VŠB – Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

Katedra informatiky

Možnosti nasazení NoSQL databázových systémů a jejich srovnání s relačními databázovými systémy

Possibilities of Deployment of NoSQL Database Systems and their Comparison with Relational Database Systems

2020 Ondřej Szkandera

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Ondřej Szkandera**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Možnosti nasazení NoSQL databázových systémů a jejich srovnání s relačními databázovými systémy
Possibilities of Deployment of NoSQL Database Systems and their Comparison with Relational Database Systems

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je analyzovat a na praktických příkladech demonstrovat nasazení NoSQL databázových systémů v reálných databázových úlohách, definovat jejich výhody a nedostatky s ohledem na nasazení, údržbu a výkon informačního systému.

1. Student se seznámí s databázovými systémy Mongo DB, hBase, Casandra a dalšími.
2. Student provede výkonnostní porovnání při budování rozsáhlých dokumentově orientovaných databázových systémů se zaměřením na rychlost tvorby databáze, manipulace se záznamy a vyhledávání, integrace do různých implementačních prostředí, zálohování, škálování a nepřetržitý provoz.
3. Student provede porovnání zvolených databázových systémů s relačními databázovými systémy a popíše výhody a nedostatky. Student posoudí jaké možnosti práce s dokumenty (JSON, XML) nabízí relační databázové systémy.
4. Bude provedena analýza nasazení na vlastní infrastruktuře, hostingu nebo cloud platformě (Azure, AWS, Google Cloud).
5. Výsledkem bude teoretická příručka, která pomůže při volbě vhodné technologie pro daný projekt s ohledem na výkon, jednoduchost nasazení a finanční náročnost.
6. Teoretické poznatky budou ověřeny na příkladu implementace rozsáhlé databáze.

Seznam doporučené odborné literatury:

[1] GORMLEY, Clinton a Zachary TONG. Elasticsearch: the definitive guide. ISBN 1449358543.
[2] SHKLAR, Leon. a Rich. ROSEN. Web application architecture: principles, protocols and practices. 2nd ed. Hoboken, NJ: Wiley, c2009. ISBN 047051860x.
[3] SHIVAKUMAR, Shailesh Kumar. Architecting high performing, scalable and available enterprise web applications. ISBN 9780128022580.
[4] WEERAWARANA, Sanjiva. Web services platform architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and more. Upper Saddle River, NJ: Prentice Hall PTR, c2005. ISBN 0131488740.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Radoslav Fasuga, Ph.D.**

Datum zadání: 01.09.2019

Datum odevzdání: 30.04.2020

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 29. dubna 2020 .....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 29. dubna 2020 .....

Rád bych poděkoval Ing. Radoslav Fasuga, Ph.D za odborné vedení v průběhu psaní této bakalářké práce.

Abstrakt

Bakalářská práce se zabývá popisem NoSQL databázových systému a jejich srovnání s relačními databázovými systémy. První část práce je zaměřena na obecným popis vlastností jednotlivých typu NoSQL databází, a jejich obecné srovnání s relačními databázovými systémy. V této části jsou také popsány základní možnosti nasazení NoSQL databází a práce s nestrukturovanými daty v oblasti relačních databázích. V druhé části práce je provedena sada výkonnostních testů, srovnávající NoSQL a relační databáze. Dále je zde ukázána práce s nestrukturovanými daty v MySQL databázi a možnost jednoduchého nasazení databázového systému MongoDB na cloudovou platformu Azure.

Klíčová slova: NoSQL databáze; Relační databáze; Nestrukturovaná data; Cache; Redis; MongoDB; MySQL; Azure

Abstract

The batchelor thesis deals with description of NoSQL database systems and their comparison with relational database systems. The first part of the thesis is focused on a general description of properties of NoSQL database types and their comparison with relational database systems. The section also describe the basic possibilities of deploying NoSQL databases and working with unstructured data in relational databases. In the second part of the thesis is performed a set of performace tests, which compares NoSQL and relational databases. It also shows work with unstructured data in MySQL database and the possibility of simple deployment of the MongoDB database system to Azure cloud platform.

Keywords: NoSQL databases; Relational databases; Unstructured data; Cache; Redis; MongoDB; MySQL; Azure

Obsah

Seznam použitých zkratek a symbolů

DML	–	Data Modification Language
HTTP	–	Hypertext Transfer Protocol
API	–	Application Programming Interface

Seznam obrázků

Seznam tabulek

1 Úvod

Cílem bakalářské práce je představit NoSQL databáze v různých situacích, pro které je vhodné jejich využití a poukázat na výhody, které z jejich použití plynou. Výsledkem práce by měla být příručka, která shrne postupy práce s NoSQL databázemi a poskytne čtenáři dostatek informací pro správnou volbu databázového systému. NoSQL databáze nabízejí odlišné možnosti než dnes tradičně oblíbené relační databáze a je proto vhodné se s těmito možnostmi seznámit. Velký důraz je kladen na porovnání výkonu jednotlivých případů nasazení s výkonem běžné relační databáze. Práce také počítá se základní znalostí čtenáře problematiky relačních databází.

V teoretické části jsou stručně popsány datové formáty, které jsou pro NoSQL velice důležité. Poté jsou popsány hlavní pojmy související s NoSQL databázemi a jejich vlastnosti, včetně kategorizace a popisu hlavních představitelů jednotlivých kategorií. Největší pozornost v teoretické části je věnovaná jednotlivým možnostem nasazení. Zde budou popsány případy, ve kterých se vyplatí využít NoSQL technologii. Mezi případy jsou zahrnuty jak situace, ve kterých NoSQL databáze pracují současně s relačními databázemi, tak i případy kdy se NoSQL použije jako primární datové úložiště. Dále jsou popsány možnosti, které nabízejí relační databáze pro práci s nestrukturovanými daty a možnosti nasazení NoSQL databází v cloudovém prostředí.

Praktická část se věnuje porovnání výkonů NoSQL databází v porovnání s relační databází, a to v různých případech nasazení, které byly popsány v teoretické části. V této části budou také provedeny výkonnostní testy, které budou použity jako podklady pro obhajobu výhod jednotlivých případů nasazení. Kromě toho zde budou ukázány praktické příklady práce s nestrukturovanými daty v databázovém systému MySQL a popis použití klientů pro některé NoSQL databáze. V poslední částí bude provedena ukázka nasazení databázového systému MongoDB na platformě Azure Cloud s použitím systému MongoDB Compass.

V závěru budou shrnuty výsledky výkonnostních testů provedené v praktické části a sumarizace závěrů, které z nich plynou.

2 Teoretická část

2.1 NoSQL databáze

Pojem NoSQL není tak jednoznačný jako je to v případě relačních databázových systémů. Mezi skupinu NoSQL databází můžeme zahrnout praktický všechny typy databází, které nejsou založeny na relačním modelu. V nejobecnější rovině pak můžeme NoSQL databáze řadit do dvou kategorií. První kategorií jsou tzv. "Pravé" (Core) NoSQL databáze. Do této skupiny spadají databázové systémy jako například MongoDB, CouchDB, Casssandra, Redis atd. Druhou kategorií v tomto členění jsou "Nepravé" (Non core) NoSQL databáze. Tady lze zařadit všechny ostatní databázové systémy, které nejsou založeny na relačním modelu. Můžou to být například objektové nebo XML databáze.

Z názvu této kategorie databází by se tedy dalo očekávat, že se jedná o jakýsi protiklad k relačním databázím. Není to však zcela pravda. Relační i NoSQL databáze vycházejí z odlišných principů, avšak určitou část mají společnou. Někdy také záleží na konfiguraci, pomocí které můžeme dosáhnout velmi podobného chování u obou typů databází. Hlavním rozdílem tedy zůstává to, pro jaké situace jsou jednotlivé typy databází nejvhodnější.

2.1.1 Formáty používané v NoSQL databázích

Jak bylo řečeno v předchozí kapitole, NoSQL není svázáno s nutností dodržení předem specifikovaného formátu dat. Data se tedy ve většině případů neukládají do tabulek, jak je tomu u relačních databází, ale používají některé datové formáty, které mají samo-popisný charakter.

Jedním z nejpopulárnějších formátu pro výměnu dat je **JSON**. Ten je založen na podmnožině programovacího jazyka JavaScript a nabízí jednoduchý způsob, jak uložit data včetně jejich struktury v jednom souboru. Princip spočívá v reprezentaci dat ve formě klič-hodnota. Pomoci tohoto formátu lze realizovat prakticky jakoukoliv datovou strukturu.

Některé databázové systémy používají upravenou verzi tohoto formátu. Úpravy většinou spočívají ve zefektivnění rychlosti při práci s daty, uložených v tomto formátu a ve velikosti potřebné pro uložení dat. Jako příklad lze uvést databázový systém MongoDB, který data ukládá pomocí formátu **BSON**. Tento typ formátu ukládá data v binární podobě a přináší oproti JSON několik výhod. Hlavní výhoda, díky které se BSON používá je jednodušší strojová zpracovatelnost oproti JSON formátu. [1]

Aby bylo možné validovat data uložena v tomto formátu existuje standard pro definování schématu JSON dokumentu. Pomocí něj lze specifikovat vzhled dokumentu, datové typy a povinné či nepovinné složky. JSON schéma lze použít pro validaci dat v rámci aplikace. Mnoho moderních NoSQL databází založených na tomto formátu již podporuje validaci oproti tomuto schématu.

Je tedy možné definovat schéma přímo v databázovém systému, díky čemuž bude docházet k automatické validaci dat. Například databázový systém MongoDB využívá a rozšiřuje standard JSON schématu. S jeho využitím lze validovat data při vkládání a aktualizacích. Lze také pro danou kolekci specifikovat úroveň dodržování daného schématu (buď budou dokumenty, které neodpovídají schématu zahozeny, nebo se pouze uloží varovná hláška o tom, že daný dokument není v souladu se specifikovaným schématem). [2]

Druhým velice často používaným formátem pro popis dat je formát **XML**. Ten ukládá samotná data mezi dvojicí značek (popřípadě jako hodnotu atributů dané značky), které reprezentují význam uložených dat. V porovnání s JSON formátem je XML pro lidi lépe čitelný, avšak při přenosu dat přes síť dochází k většímu časovému zpoždění z důvodu větší velikosti metadat.

Stejně tak jako JSON, tak i XML podporuje validaci dokumentu. Validace XML dokumentu je dokonce více rozšířená, než je tomu u formátu JSON. Existují dva základní způsoby, jak takovou validaci provádět. První možností je DTD (Document Type Definition). Pomoci několika elementů lze specifikovat vzhled XML dokumentu a povinné a nepovinné elementy. Nevýhodou tohoto přístupu je malá rozšiřitelnost, neschopnost specifikovat datové typy a specifická syntaxe. Tyto nevýhody se snaží kompenzovat druhý způsob validace, čímž je XSD (XML schema definition). Jedná se o XML dokument, který slouží pro popis XML dokumentu reprezentující data. Je snadno rozšiřitelný, podporuje vlastní datové typy a jeho syntaxe vychází z jazyka XML. Z těchto důvodu se jedná o nejpoužívanější způsob validace XML dokumentu. [3] [4]

Příkladem systému, který využívá tento formát pro ukládání data je například eXist-db.

2.1.2 CAP teorém

Pojem CAP teorém byl poprvé použit Ericem Brewerem a je úzce svázán s distribuovanými databázovými systémy, mezi které se řadí i NoSQL databáze.

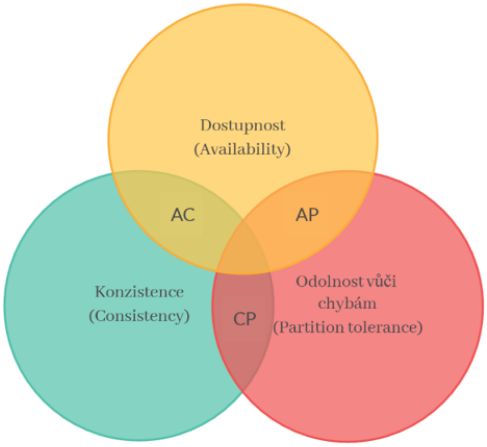
Žádný distribuovaný databázový systém není schopen sám o sobě zaručit, že nedojde k výpadku části sítě, která by zapříčinila jeho nedostupnost. Abychom zabránili výpadku, je proto nutné přistoupit k rozdělení sítě. Pokud pak v části sítě dojde k výpadku některého uzlu, ostatní uzly jsou nadále schopny přijímat požadavky a poskytovat data. Tento princip je označován jako **odolnost vůči chybám** (z anglického partition tolerance) a je zastoupen písmenem P v akronymu CAP. Tvrzení CAP teorému pak říká, že v distribuovaných databázových systémech se při výpadku části sítě musíme vždy rozhodnout mezi **dostupností** (z anglického availability) zastoupeného písmenem A, a **konzistencí** (z anglického consistency) zastoupeného písmenem C a nelze dosáhnout obou těchto záruk současně. [5]

Konzistence znamená, že pří každém čtení obdržíme vždy nejaktuálnější data nebo chybovou hlášku. V praxi to znamená, že dotaz na jakýkoliv uzel v clusteru[1] vrátí pokaždé stejná

[1] Cluster je označení pro sadu dvou a více zařízení, které mezi sebou spoluracují pro dosažení určitého cíle.

(konzistentní) data.

Dostupnosti se myslí schopnost systému úspěšně odpovědět na všechny dotazy i za předpokladu, že data, které vrátí, nejsou nejaktuálnější. Toto nastává například v případě asynchronní replikace, kdy jednotlivé servery v clusteru obsahuji odlišnou množinu dat. V takovém případě je možné, že klient obdrží odlišné odpovědi na stejný dotaz a to do té doby, než dojde k synchronizaci všech uzlů v clusteru.



Obrázek 1: CAP teorém

NoSQL databáze tradičně nabízejí vlastnost odolnosti vůči chybám. K této vlastnosti lze pak doplnit vlastnost konzistence nebo dostupnosti. Relační databáze pak ve většině případů odpovídají kombinaci písmen AC a tedy dostupností a konzistence.

CAP teorém lze považovat za jakési kontinuum na jehož základě jsou BASE a ACID na opačných koncích. Zatímco se ACID zaměřuje na splnění vlastností dostupnosti a konzistence, BASE je zaměřen spíše na splnění dostupnosti a odolnosti vůči chybám. BASE se tedy podmínky ACID nesnaží nahradit, ale spíš doplňuje jejich protipól.

2.1.3 ACID a BASE

Na rozdíl od relačních databází jsou NoSQL databáze oproštěny od nutnosti dodržování transakční logiky a podmínek ACID. Místo něj se uchytila zkratka BASE, která odpovídá principům NoSQL databází daleko lépe.

Zkratka BASE vychází ze 3 anglických sousloví. První dvě písmena **BE** označují pojem **Basically available**. Tato vlastnost říká, že systémy vyhovující této podmínce budou zaručovat dostupnost z hlediska CAP teorému. Jinými slovy je pro ně nezbytné, aby zaručovaly vlastnost odolnosti vůči chybám a k této vlastnosti si doplnily jednu ze dvou zbývajících. Písmeno **S** reprezentuje další vlastnost označenou jako **Soft state**. Její význam je pro distribuované systémy poměrně typický a znamená, že data v systému nemusí být vždy zcela konzistentní. Vezměme

například v úvahu situaci, kdy jsou v clusteru obsaženy 2 uzly. Každý z uzlů může provádět všechny CRUD operace (Create, Read, Update, Delete) a mezi oběma uzly probíhá replikace dat. Uživatel A komunikuje s prvním uzlem a uživatel B s uzlem druhým. Uživatel A provede změny na určité množině dat, které se bude uživatel B snažit přečíst. V takovém případě bude docházet k tomu, že uživatel B bude dostávat neaktuální informace až do doby, než se provede replikace všech změněných dat. V praxi je prodleva mezi jednotlivými časy synchronizací poměrně malá, nicméně je tato možnost nekonzistence u takto postavených systému stále možná. Vlastnost, zahrnuta pod písmenem **E**, se dá chápat jako dovětek k předchozí vlastnosti. Tato vlastnost vyjadřuje schopnost systému dostat se po určitém čase do konzistentního stavu. Toto chování lze pozorovat i v našem příkladě. Systém se nacházel v nekonzistentním stavu až do chvíle, než došlo k replikaci. [6]

ACID	BASE
Silná konzistence	Slabá konzistence
Izolovanost	V první řadě dostupnost
Orientace na potvrzení (commit)	Přístup „dělám co můžu“
Vnořené transakce	Přibližné odpovědi jsou dostačující
Menší dostupnost	Rychlé
Pesimistický přístup	Optimistický přístup
Náročné rozšíření	Jednoduché rozšíření

Tabulka 1: Shrnutí rozdílů mezi ACID a BASE

2.1.4 Kategorie NoSQL databází

Databáze typu klíč-hodnota

Princip těchto databází je založen na párech typu klíč a hodnota, kdy jeden klíč ukazuje vždy pouze na jednu hodnotu. Princip je velmi podobný asociativnímu poli nebo slovníku (dictionary). Mapování probíhá jednosměrně od klíče k hodnotě.

Většina databázových systémů tohoto typu nepodporuje žádný dotazovací jazyk jako je to například u relačních databázových systémů. Z tohoto důvodu nelze provádět dotazy nad hodnotami ale pouze nad klíčem. Pro získání hodnoty je tedy nutná znalost klíče, který na danou hodnotu ukazuje.

Tento typ databází lze považovat za nejjednodušší a nejzákladnější princip, jak přistupovat k ukládání dat.

Dnes nejpoužívanějším a nejznámějším představitelem tohoto typu databáze je bezesporu **Redis**. Jedná se o open-source databázový systém, podporující mnoho různých datových struktur (řetězce, listy, množiny, seřazené množiny, hashe). Redis je označován jako in-memory databáze, což by mohlo vést k domněnce, že se nejedná o perzistentní úložiště. Ve skutečnosti však stojí někde mezi perzistentním a neperzistentním typem úložiště. Redis převážně uchovává data v primární paměti. Díky tomu poskytuje velmi vysoký výkon. Obsah primární paměti je pak na základě konfigurace ukládán na disk, čímž je docílena perzistence dat. V případě výpadku pak nepřicházíme o uložená data (maximálně o pár posledně uložených záznamů).

Dalšími představiteli může být například Memcached, Riak, Berkeley DB, Level DB, Amazon Dynamo DB.

Dokumentové databáze

Dokumentová databáze je označení pro typ nerelačního databázového systému, který ukládá data v souborech (dokumentech) pomocí jednoho ze samo-popisných formátů (viz. kapitola 2.1.1). Jeden záznam je označován jako dokument. V dokumentu jsou uloženy jednotlivé záznamy, které mohou nabývat různých datových typů. Dokumenty, které udržují data stejného typu se organizují do kolekcí. Sada kolekcí je pak udržovaná v samotné databázi. V následující tabulce je porovnání terminologie používané v dokumentově orientovaných databázích vzhledem k relačním databázím. [7] [8]

Dokumentová databáze	Relační databáze
Databáze	Databáze
Kolekce	Tabulka
Dokument	Řádek
Záznam (atribut)	Sloupec

Tabulka 2: Srovnání terminologie dokumentové a relační databáze

V dnešní době existuje mnoho různých implementací dokumentových databází. Jednou z nejpoužívanějších technologií tohoto typu je **MongoDB**. Jedná se o multiplatformní databázový systém vyvíjený pod open-source licenci, který byl vydán v roce 2009 a nabízí jednoduchý a rychlý způsob pro ukládání velkých objemů dat. Data jsou ukládána ve formátu BSON, každý dokument může mít odlišný formát a velikost. MongoDB také nabízí rozšířené možnosti jako například kontrolu struktury jednotlivých dokumentů.

Mezi další databázové systémy tohoto typu patří například Apache CouchDB, CouchBase server, MarkLogic, InterSystems nebo OrientDB.

Sloupcově orientované NoSQL databáze

Sloupcově orientované databáze se na první pohled velmi podobají klasickým relačním databázím. Je mezi nimi ovšem velký rozdíl. Tím největším rozdílem je princip, jakým sloupcově orientované databáze ukládají data. Na rozdíl od relačních databází, které jsou řádkově orientované, sloupcově orientované databáze ukládají každý definovaný sloupec jako souvislou sekvenci dat. Tento způsob uložení dat přináší velkou výhodu při provádění agregačních funkcí. [9]

Vezměme například situaci, ve které máme ve sloupci A uloženy ceny produktu a v ostatních sloupcích další relevantní informace o produktu a chceme spočítat jakou cenu mají všechny výrobky dohromady. V případě řádkově orientovaných databází je ve většině případů nutné projít všechny sloupce i řádky postupně. U sloupcově orientovaných databází však stačí aby databázový systém prošel pouze ten sloupec, který má pro nás relevantní data (tedy sloupec A). Hlavní rozdíl mezi relačními a sloupcově orientovanými databázemi je tady především ve fyzickém uložení dat na disku.

Představitelem tohoto typu databázových systému je například **Apache Cassandra**. Jedná se o masivně škálovatelný a vysoce výkonný distribuovaný databázový systém který je navržen pro zpracování velmi velkého objemu dat. Cassandra byla vyvinuta v roce 2011 společnosti Facebook a od této doby se začala velmi rychle rozvíjet a prosazovat na poli zpracování vysoko objemných dat.

Dalšími představiteli tohoto typu databází je například Apache HBase, MonetDB, CrateDB nebo ClickHouse.

Grafově orientované databázové systémy

Tento typ databázových systému je poměrně odlišný od těch ostatních. Základ pro tyto databáze jsou uzly a vztahy. Uzly reprezentují entity a vztahy určují v jakých asociacích se entity nacházejí. Každý uzel obsahuje seznam. Položky tohoto seznamu reprezentují vztahy mezi uzly.[10]

Grafově orientované databáze se využívají méně častěji než jiné typy databází. Je to zapříčiněno tím, že je tento typ databází vhodný pouze pro specifickou problematiku. Jsou vhodné především pro rozsáhlá vztahově orientovaná data. Při ukládání správného typu dat nabízejí tyto databázové systémy velmi vysoký výkon, kterému relační databáze nemohou konkurovat.

Představitelé tohoto typů databází jsou například Neo4j nebo JanusGraph.

2.2 Porovnání SQL a NoSQL databází

Relační databáze, často také označovány jako SQL databáze, se od NoSQL databází liší v mnoha směrech. Hlavním aktérem, který vytváří rozdíly mezi těmito technologiemi je právě jazyk SQL primárně využívaný relačními databázovými systémy. Tento jazyk poskytuje velmi efektivní

způsob práce s daty. Na druhou stranu klade určitá omezení vztahující se k podobě uložených dat.

SQL databáze ukládají data ve formě tabulek, které reprezentují entity skutečného světa. Sloupce tabulky představují atributy (vlastnosti) entity a řádky reprezentují konkrétní výskyty dané entity. Mezi tabulkami lze vytvářet vazby na základě skutečných vztahů mezi entitami. Pro vytváření tabulek a jejich vazeb existují pravidla tzv. normální formy. Tyto normální formy mají za cíl vytvoření smysluplných logických vazeb a eliminaci redundantních dat v databázi. Tabulka splňující pravidla normálních forem se označuje jako normalizovaná tabulka. [11] Díky této struktuře je databázový systém schopen řídit integritu dat. K tomuto jsou definovaná určitá integritní omezení, která jsou ověřovaná při vykonávání téměř každého SQL dotazu. Všechny relační databáze vycházejí z tohoto principu ukládání dat. Díky tomu, že se všechny implementace relačních databází řídí téměř stejnými pravidly (existuje norma, která však není vždy zcela dodržena), jsme schopni využít možností jazyka SQL napříč mnoha různými relačními databázovými systémy téměř totožným způsobem.

Na druhou stranu, v případě NoSQL databází, je situace velmi odlišná. Existuje několik různých typů databází a každý tento typ ukládá data odlišným způsobem. Dokonce existují odlišné přístupy i mezi databázemi stejného typu. Například MongoDB a CouchDB jsou databázové systémy spadající do kategorie dokumentově orientovaných databází. Oba tyto systémy mají však naprosto odlišný způsob komunikace. MongoDB komunikuje na základě proprietárního protokol [12], založeného na soketech a síťovém protokolu TCP a CouchDB poskytuje API rozhraní [13], jež je přístupně přes standardní protokol HTTP. Kvůli odlišným způsobům přístupů je nereálné vytvořit jeden unifikovaný jazyk pro manipulaci s daty napříč různými NoSQL systémy jako to nabízí jazyk SQL. Toto je oproti relačním databázím velká nevýhoda. Zatímco přechod z jedné relační databáze na jinou je poměrně jednoduchý, u NoSQL databází to ve většině případů znamená velký zásah do aplikace. Některé NoSQL databázové systémy nabízejí možnost dotazování pomocí SQL jazyka. Nejde však o využití klasického jazyka SQL, ale pouze o usnadnění zápisu dotazu. V rámci vnitřní logiky databázového systému se takto napsané dotazy převedou do podoby dotazovacího jazyka přirozeného pro daný databázový systém.

Jazyk SQL je velice mocný nástroj, který posouvá možnosti relačních databázích na daleko vyšší úroveň. Všechny relační databáze jsou postaveny na principech ACID a jsou tedy vázány transakční logikou. I přesto, že lze transakční zpracovávání určitým způsobem optimalizovat, například snížením úrovně izolovanosti, jde o zásadní vlastnost, která výrazným způsobem ovlivňuje rychlost databáze. Platí, že čím je úroveň izolovanosti nižší tím je rychlost zpracováváni transakce vyšší. Snižování úrovně izolovanosti však zvyšuje problém s nekonzistencí dat. Hlavně z tohoto důvodu jsou NoSQL databáze obecně považovány za rychlejší.

Díky jazyku SQL jsou relační databáze vhodnější pro komplexnější dotazy, zatímco NoSQL databáze jsou kvůli absenci sofistikovanějšího dotazovacího rozhraní na vyšší úrovni vhodnější

spíše pro jednodušší dotazy. Samozřejmě to neznamená, že nelze vykonávat složitější dotazy nad databází i v NoSQL databázových systémech. Dotazy jsou však daleko komplikovanější než v případě SQL databází a často je nutné je řešit na aplikační úrovni.

Druhou doménou, ve které NoSQL databáze vynikají, je jejich škálovatelnost. Zatímco relační databázové systémy se škálují výhradně vertikálně (scale-up), NoSQL databáze nabízejí možnost horizontálního škálování (scale-out). U vertikálního škálování dochází pouze k navyšování hardwarové kapacity zařízení, na kterém je databázový systém spuštěn. Horizontální škálování dovoluje rozložit zátěž mezi několik oddělených zařízení, které společně tvoří tzv. **cluster**. Výhodou je snížení investic do stále výkonnějšího hardwaru. Namísto toho se v případě potřeby přidá do clusteru další zařízení, jež nemusí být tak výkonné. S horizontálním škálováním je spojen také anglický pojem **sharding**. Ten označuje techniku, při které se data rozloží napříč clusterem. Každý uzel v clusteru pak udržuje specifickou množinu dat na základě předem specifikovaného klíče. Horizontální škálování je pro databázové systémy založené na relačním modelu velmi náročnou disciplínou. Existují určité hybridní způsoby, jak docílit podobného chování. Každý tento způsob však přináší určité problémy, které je potřeba vyřešit. Systém se kvůli tomu velice komplikuje a často také dochází k určitému poklesu výkonu.

Dalším podstatným rozdílem je úroveň podpory. Pro relační databáze existuje velice široká podpora napříč mnoha programovacími jazyky. Kromě podpory na úrovní základní komunikace existuje také řada frameworků, které velice ulehčují práci při vývoji. Mezi nejrozšířenější frameworky patří například **Entity framework** nebo **Hibernate**. Nesourodé komunikační prostředky činí stejný problém pro NoSQL databáze podstatně komplikovanějším. Některé z databázových systému poskytují knihovny pro komunikaci s databází, u ostatních databází je potřeba využít knihovny třetích stran nebo jiné prostředky. Tento problém není neřešitelný, ale podstatně komplikuje a prodlužuje vývoj aplikace. Zastoupení frameworků pro tyto technologie je prakticky nulové, a proto je nutné vytvářet vrstvu přístupu k datům většinou manuálně.

2.3 Rozdíly v možnostech práce mezi SQL a NoSQL databázemi

Již v předchozí kapitole bylo naznačeno, že jsou způsoby práce s NoSQL databázemi velmi různorodé. Různorodost je zapříčiněna odlišnými koncepty ukládání dat a způsoby komunikace. Oproti tomu mají relační databáze jeden standardizovaný způsob manipulace s daty, který zastřešuje standard jazyka SQL. Každá implementace relační databáze má určité specifické rysy v implementaci standardu SQL. Implementace jádra standardu však zůstává u většiny implementací dodržena.

Každá NoSQL technologie nabízí jiné možnosti práce s daty. Operace jako vkládání, aktualizace a mazání dat je nutnou součástí všech databází. Implementace těchto operací se tedy liší zejména na úrovni použitého datového modelu. Největším rozdílem tedy zůstávají možnosti při dotazování dat.

Pokud se zaměříme na databáze typu klíč-hodnot, pak zjistíme, že jsou možnosti dotazování velice omezené. Tato omezení vycházejí z podstaty tohoto typu databází. Možnosti dotazování se tedy zúží pouze na operaci `GET`, která na základě klíče vrátí požadovanou hodnotu. V případě, že chceme výstupy nějakým způsobem filtrovat, je nutné tomu přizpůsobit datový model (viz kapitola 2.5), nebo filtrování realizovat na úrovni aplikace. Některé systémy nabízejí rozšířené možnosti práce. Některé databázové systémy jako je například Redis poskytují rozšířené možnosti pro ukládání dat. Pak se může množina operací nad systémem rozšířit o některé další operace jako je například `POP` pro odebrání prvku z pole nebo například `GetRange` pro získání určitého rozsahu z listu.

Způsob práce se sloupcově orientovanými databázemi je velice podobná práci s relačními databázemi a jazykem SQL. Mnoho implementací sloupcově orientovaných databází vychází ze stejné syntaxe jako má jazyk SQL. Hlavním rozdílem těchto databází oproti relačním databázím je ve vnitřní struktuře uložených dat. Kvůli odlišnostem ve vnitřní struktuře vznikají i odlišnosti ve vykonávání dotazů. I přesto, že dotazy vypadají velice podobně mnoho operací se chová velmi odlišně. Například dotazy s agregačními funkcemi (průměr, suma, ...) budou ve sloupcových databázích vykonány velice efektivně, zatímco operace spojení tabulek (JOIN) bude mít ve většině případů velice negativní vliv na výkon[2].

Databáze typu klíč-hodnota mají velmi omezené možnosti dotazování, zatímco v případě sloupcově orientovaných databázích lze dosáhnou podobně komplexních dotazů jako v relačních databázích. Dokumentové databáze stojí někde uprostřed mezi těmito úrovněmi. Samozřejmě záleží na tom, s jakou implementací dokumentové databáze pracujeme. Každá nabízí jinou množinu operací, které lze při dotazech využít. V příloze C jsou uvedeny ukázky příkazu jazyka MongoDB, u kterých jsou uvedeny ekvivalentní příkazy jazyka SQL.

2.4 Použití NoSQL databází v roli cache serveru

Cache paměti se nejčastěji využívají jako první krok při zvyšování výkonu aplikace. Princip spočívá v tom, že mezi dvě vrstvy aplikace, které mezi si mezi sebou předávají data vložíme mezivrstvu, která umí pracovat s daty efektivněji. Ve většině případů se jedná právě o vrstvu aplikační a o vrstvu databázovou, mezi které se přidá vyrovnávací paměť (cache server).

2.4.1 Typy cache pamětí

Existuje několik úrovní cache paměti. Každá z nich má určité své výhody i nevýhody. V určitých situacích jsme také limitování technologii, kterou používáme. Každá technologie nabízí určitou úroveň možnosti práce s cache paměti. V následujících odstavcích jsou popsány hlavní typy cache pamětí.

[2]Operace JOIN obecně způsobuje výkonnostní problémy. Relační databáze, které ukládají data ve formě řádků jsou však na tuto operaci daleko lépe uzpůsobeny než sloupcově orientované databáze.

Cache integrovaná do databáze

U některých databázových systémů existuje zabudovaná cache paměť. Tento druh cache paměti poskytuje možnost využití write-throught strategie (viz. 2.4.2) bez nutnosti přizpůsobovat vlastní implementaci aplikace. O naplnění cache paměti se stará sám databázový systém. Data uložená v cache paměti jsou převážně vždy konzistentní. Nevýhodou je to, že je taková cache paměť často limitovaná velikostí dostupné paměti, která je pro ni určena. Integrovanou cache paměť obsahuje například MySQL nebo MongoDB.

Lokální cache

Lokální cache ukládá nejčastěji přistupovaná data do lokální paměti. Tento přístup je rychlejší než ostatní, protože nedochází ke zpoždění při přenosu dat přes síť. Hlavní nevýhodou je pak to, že každá instance aplikace využívá svou vlastní cache paměť. Dnešní systémy jsou však postaveny tak, aby bylo možné využívat několik aplikačních serverů. Z tohoto důvodů je ve většině případů žádoucí, aby byla data v cache paměti sdílená mezi všemi instancemi aplikace. Jako lokální cache můžeme považovat například Session u webových aplikaci.

Vzdálená cache

Vzdálená cache funguje tak, že jedna nebo více instancí cache serveru běží odděleně od databázového serveru většinou i na odděleném zařízení. Ve většině případů (ne však podmínkou) jsou tyto systémy postaveny na databázových systémů typu klíč-hodnota jako je například Redis nebo Memcached.

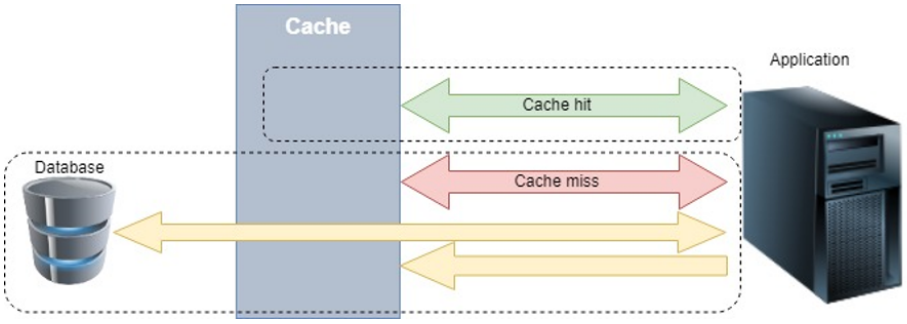
I přesto, že dochází k částečné degradaci výkonu kvůli nutnosti přenést data přes síť, tak se průměrná doba dotazu na vzdálený server pohybuje v řádech několika milisekund. V porovnání s časem, který je potřebný k přístupu na disk je doba přenosu dat sítí zanedbatelná.

Při nasazování cache serveru je nutné zvážit mnoho faktorů a na jejich základě zvolit správnou strategii [14] [15], kterou použijeme. Jedním z hlavních faktoru ovlivňující výběr je to, zda jsme ochotní dopustit nekonzistentní stav mezi primární databází a cache paměti. Dalším důležitým faktorem je efektivita využívání paměti vyhrazené pro cache. V neposlední řadě je také potřeba zvážit, jaké informace chceme v cache paměti uchovávat. Všechny tyto faktory výrazně ovlivňují použitou strategii a způsob jakým se bude cache paměť chovat.

2.4.2 Cache strategie

Nejjednodušší způsob implementace cache paměti se nazývá **Lazy load**. Princip spočívá v tom, že pokud aplikace potřebuje získat data, zkontroluje nejprve cache paměť. Pokud jsou požadovaná data v cache paměti nalezena (cache-hit), aplikace vrátí uživateli data přímo z této pamětí. Pokud požadovaná data v cache paměti nalezena nejsou (cache-miss), aplikace se dotáže na primární databázový server, ten vrátí požadovaná data, která jsou následně uložena do cache pamětí a vrácena uživateli.

Tento přístup je výhodný z hlediska paměti, kterou cache využívá, protože se v paměti udržují pouze ty data, která byly dotazovány. Další výhodou je velmi jednoduchá implementace. Důležitým aspektem této strategie je také možnost ukládat taková data, která nemají stejnou datovou strukturu jako je datová struktura v primární databázi. Vezměme například situaci, ve které provozujeme webový server a chceme do cache paměti ukládat výsledky nejčastěji pokládaných dotazů. V takovém případě by jako klíč mohla být použita URL adresa požadavku a jako hodnota by se ukládal výstup z databáze převedený do jednoho ze samo-popisných formátů. Jakmile bychom tuto strategii nasadili, opakované dotazy na webový server by se načítali přímo z cache paměti a nebylo by nutné zatěžovat primární databázi. Je zde ovšem problém s tím, že není jednoduché sledovat, které data těmto dotazům odpovídají. Pokud by tedy došlo ke změně dat, uložených v primární databázi, cache paměť by stále vracela původní data. Je zde tedy riziko nekonzistentního stavu mezi primární databází a cache paměti.



Obrázek 2: Znázornění průběhu dotazu strategie Lazy load

Problém nekonzistentního stavu se dá částečně redukovat použitím**TTL** (Time to Live) atributu. Ten zaručí to, že budou data po uplynutí časového intervalu prohlášená za nevalidní a cache server již nebude vracet odpověď na požadavky s takto označenými daty. Správná volba velikosti TTL parametru je při použití této strategie zásadním faktorem. Při zvolení nízké hodnoty TTL bude konzistence mezi primární databází a cache paměti větší, než při použití větší hodnoty ale o to více budou nastávat situace kdy může dojít ke „cache-miss“ a rychlost se bude degradovat.

Další často používaná strategie se nazývá **Write-throught**. Tento typ strategie obrací postup, jakým je cache paměť naplňována. Zatímco v případě Lazy load strategie jsou data do cache paměti vkládaná až tehdy, kdy jsou skutečně dotazována, v případě Write-through strategie jsou data ukládány do cache paměti souvisle s primární databází.

Ve většině případů se tato strategie implementuje společně s Lazy load strategií, a to především při procesu prvotního naplnění cache paměti. Data, která však nebyla do paměti načtena musí být získaná z primárního uložiště jak při čtení, tak i při jakékoliv modifikaci.

Hlavní nevýhodou je plýtvání paměťového prostoru určeného pro cache paměť. Do paměti se ukládají všechna data, bez ohledu na relevanci jejich dotazování. Může se tak stát, že některá

data uložena v cache paměti se nedotazují vůbec a zbytečně tak zabírají paměť. Tento problém se dá vyřešit přidáním TTL parametru. Tím docílíme toho, že se data, které není potřeba v cache paměti uchovávat po čase prohlásí za nevalidní a následně se z paměti smažou. Další nevýhodou je poměrně složitá implementace a zpomalení při zápisu a aktualizací dat. Je to způsobeno tím, že při každé operaci modifikující data je nutné provést kontrolu, zda cache paměť obsahuje odpovídající data. V případě, že paměť data neobsahuje, je nutné provést zápis změn do primární databáze a následně změny uložit i do cache paměti.

Klíčovým problémem tohoto přístupu je také nutnost zaručit jednoznačné mapování mezi primární databází a cache paměti. Toto je ve většině případů zaručeno tím, že klíč jednoho záznamu v cache paměti je totožný s klíčem v primární databází ukazující na stejná data.

Hlavní myšlenkou, jež je zároveň jednou z největších výhod tohoto přístupu, je téměř úplná konzistence mezi primární databází a cache paměti. Toto chování se částečně ztrácí při použití TTL parametru pro invalidaci málo používaných dat. Využití toho parametru lze v tomto případě částečně optimalizovat. Zatímco v případě strategie Lazy load bylo nutné pro udržení konzistence dat vždy provést invalidaci dat po uplynutí doby TTL parametru, v případě Write through strategie lze parametr TTL obnovit vždy, při každém dotazu na požadovaná data. Často dotazovaná data tak budou stále aktuální a vyhneme se i častým přístupům do primární databáze.

V praxi se pak často setkáváme s kombinací obou těchto strategii a jejich modifikacemi. Pro určitá data tak můžeme zvolit strategii jinou než pro ty ostatní. Mezi modifikace pak můžeme zařadit například takové systémy, které využívají strategii Write-throught, ale data se do primární databáze zapisují až po specifikované době. Takovému přístupu se často říká **Write back** a je vhodný pro situace, ve kterých zapisujeme do databáze velké množství dat. Jednou z dalších často vyžívaných modifikaci je přístup, ve kterém se využívá Lazy load strategie s využitím TTL, ovšem namísto toho, aby se data do cache paměti naplňovali až tehdy, jakmile jsou dotázány, dochází k tomu, že jsou data v určitých intervalem automaticky v cache paměti obnovována bez toho, aby byla vyvolána nějaká vnější akce. Všechny tyto modifikace jsou však poměrně náročné na implementaci.

Vzhledem k povaze NoSQL databází, a především pak databází typu klíč-hodnota, je vhodné o nich uvažovat právě v souvislosti s cache paměti. Jejich použití je poměrně jednoduché a rychlé. Na rozdíl od obyčejných cache pamětí, nám databázové systémy jako je **Redis** nabízejí mnohem více pokročilejších datových typů, které poskytují daleko jednodušší práci s daty, které nelze jednoduchým způsobem převést do podoby klíč-hodnota. Redis také nabízí možnost perzistentního uložení velkého objemu dat, které by v jiných případech (cache integrovaná do databáze, lokální cache) bylo jen těžko dosažitelné. Při využití NoSQL databází jako cache paměť nám také vzniká mezivrstva mezi aplikací a primární databází. Tato vrstva je velmi dobře škálovatelná, což nám umožňuje jednoduché navýšení jejího výkonu v případě potřeby.

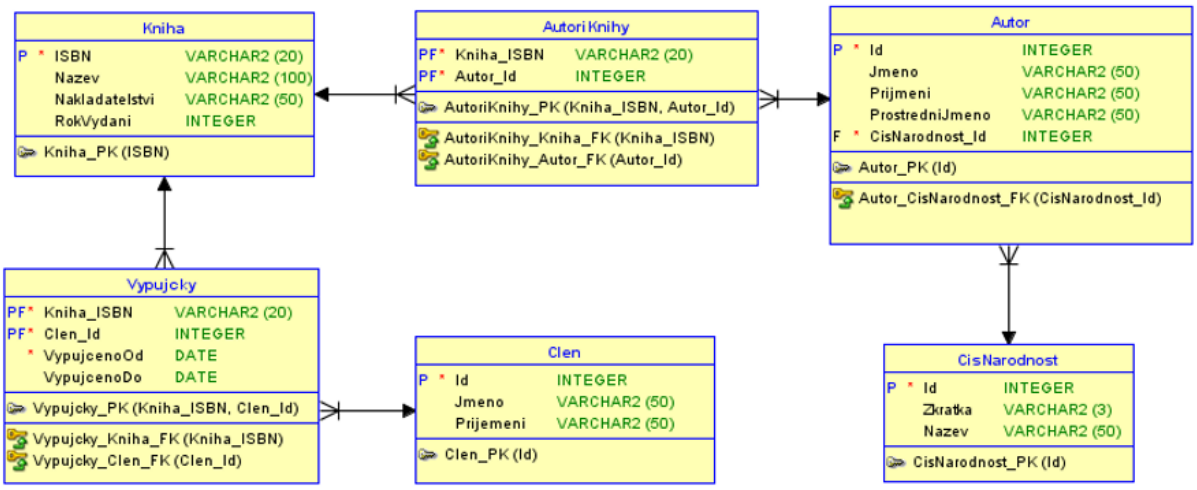
Pokud se rozhodneme pracovat s NoSQL databází jako s cache paměti pro primární databázové úložiště, je nutné brát v úvahu fakt, že se ve většině případů nejedná o kompatibilní systémy, které by mohli mezi sebou komunikovat bez prostředníka. Proto je nutné logiku dotazováni řešit na úrovni aplikace. Z tohoto důvodu je nutné již na začátku projektu zvážit, jakým způsobem bude cache paměť využívána. Pokud se vrstva cache paměti přidává do již existující aplikace, která na to není připravena, je situace poměrně složitá a ve většině případů vyžaduje rozsáhlý refaktoring aplikace.

2.5 NoSQL v roli primárního datového uložiště

Samozřejmě lze o NoSQL databázích uvažovat jako o primárním perzistentním úložišti. Tento přístup však s sebou přináší trochu jiný pohled na samotné data a jejich zpracovávání, než tomu bylo u relačních databází. Jak už bylo zmíněno výše, relační databáze mají strukturu dat danou poměrně jasně a při řešení jednoho problému se dospěje v každém případě k velmi podobnému výsledku. Toto není u NoSQL databází zcela pravda a záleží na konkrétním pohledu na data a jejich zpracování. Existuje také několik typů NoSQL databází, jejichž přístup k datům je velmi odlišný. Toto chování je pro relační databáze také nezvyklé. I přesto, že existuje mnoho implementací databázových systému založené na relační algebře, všechny pracují s daty podobným způsobem a liší se tím, jak jsou konkrétní postupy a funkce implementovány.

Způsoby uložení dat v MongoDB

MongoDB uchovává data v dokumentech. Každý dokument reprezentuje jeden záznam v databázi. Dokumenty v sobě uchovávají položky se samotnými daty. Položky mohou být skalárních datových typů, polí nebo objektů. Právě možnost uložení dat ve formě pole nebo objektu přináší rozšířené možnosti, jak data ukládat.



Obrázek 3: Doménový model pro ukázku uložení dat v MongoDB

ER diagram na obrázku 3 reprezentuje schéma uložení dat tak, jak by tomu bylo v případě relační databáze. Toto schéma bude používáno pro všechny ukázky v této kapitole. V databázi se tedy budou uchovávat základní informace o knihách, jejich autorech a členech knihovny. Ke každému autoru je pak přidána informace o jeho národnosti. Tabulka výpůjčky uchovává informace o výpůjčkách knih jednotlivých členů. Pro ukázky budou vždy použity pouze části tohoto diagramu, na kterém bude daná problematika vysvětlena.

Jedna z velkých nevýhod relačních databázových systému je neschopnost uložení relace typu M:N přímo a je nutné vytvářet vazební tabulku. Při dotazech je pak nutné využívat konstrukce JOIN. MongoDB nám však nabízí několik možností jak relaci M:N vytvořit. První možnost je ta, že se necháme inspirovat přístupem relačních databází a vytvoříme jeden dokument navíc, který budou plnit stejnou funkci jako vazební tabulka. Uložení dat Knih, Členů a jejich výpůjček by mohl vypadat tak, jak je to znázorněno na obrázku 4.

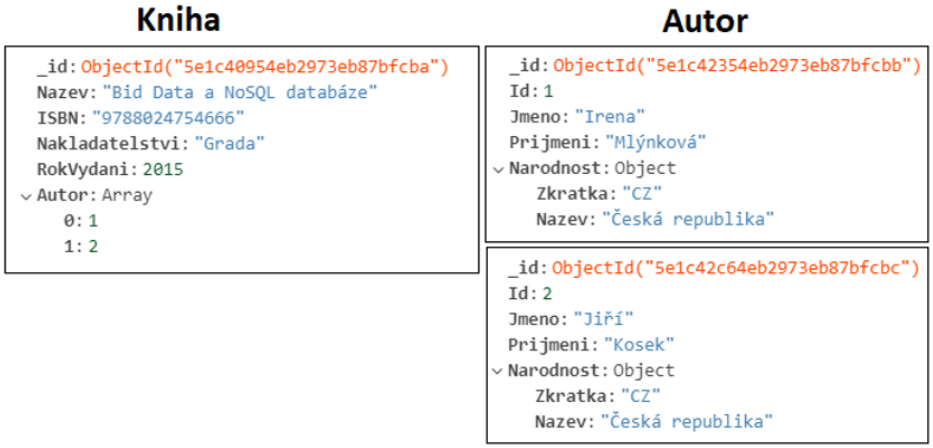
Zde můžeme vidět řešení situace prvním způsobem, který odpovídá tomu, jak by se stejný problém vyřešil v relační databázi. Tento způsob je pro dokumentové databázové systémy obecně velmi nevhodný. Zatímco SQL databáze obsahují konstrukci JOIN, která zvyšuje efektivitu a přehlednost práce při spojování tabulek, dokumentové databáze tuto funkcionalitu nenabízejí vůbec, anebo jen v omezené míře. Pokud bychom tedy chtěli na výstupu dotazu informace o knize včetně autorů, museli bychom provést spojení 3 kolekcí, anebo provést 3 samostatné dotazy.

Obrázek 4: Reprezentace vztahů pomoci vazebního dokumentu

Kromě ukázky relace N:M, lze v tomto příkladě vidět realizaci vazby 1:N a to v případě informací o národnosti autora. Abychom se vyhnuli problémům s normálními formami, ukládáme dodatečné informace o národnosti autora v samostatné tabulce. U dokumentově orientovaných databází žádné normální formy nemáme, a proto si můžeme dovolit uložení informací o národnosti jako vnořený objekt přímo do dokumentu obsahující informace o samotném autorovi. Tento přístup je výhodný z hlediska počtu dotazů na databází, kdy nám stačí provést pouze jeden dotaz nad jednou kolekcí a získáme všechny informace, které potřebujeme. Nevýhodou je

pak vznikající duplicita a z toho plynoucí náročnost na množinovou úpravu těchto dat. Pokud by naše aplikace obsahovala 100 autorů z jedné země, pak by při změně jedné informace o dané zemi muselo být provedeno 100 změn v databázi, zatímco v případě relační databáze by to byla pouze jedna změna. [16]

Další možnost, jak pracovat se vztahy v dokumentově orientovaných databázích je využití jejich schopností ukládat data i do komplexnějších datových struktur. Použití těchto datových struktur nám zredukuje počet kolekcí podle toho, jaký datový typ se rozhodneme použít. První datovou strukturou vhodnou pro tento účel je klasické pole dat. Vytvoření tohoto typu vazby funguje tak, že se do dokumentu přidá pole, jehož hodnoty budou odkazovat na jedinečný identifikátor v dokumentu jiném. Na obrázku 5 je uveden příklad, který reprezentuje stejnou datovou strukturu jako předchozí příklad.



Obrázek 5: Reprezentace vztahu použitím pole hodnot

Jak lze vidět z ukázky, tak se nám snížil počet kolekci, tím že jsme odstranili vazební tabulku. Pro získání dat o knize včetně dat o autorovi se nám tedy snížil počet dotazu na dva. Tento přístup je výhodný tehdy, když dochází k častým změnám odkazované kolekce. Změny jsou provedeny pouze na jednom místě. Nevýhodou je fakt, že musíme provést nejprve dotaz nad první kolekcí a podle získaných dat procházet kolekci druhou, což je velice neefektivní. MongoDB sice nabízí možnost spojování tabulek pomocí příkazu `lookup`. Tento příkaz však má svá omezení a v mnoha případech se raději použije alternativa dvou dotazů.

Další datovou strukturu, kterou lze pro tento účel využít je objekt. Jeho použití je naznačeno již v prvním příkladě, kde se jednalo o relaci typu 1:N, kdy jedná národnost mohla náležet více autorům. Pro úplnost je v následujícím příkladu naznačena kombinace obou postupů (využití pole a objektu) pro reprezentaci vazby v rámci jednoho dokumentu.

Touto kombinací postupů docílíme toho, že budou všechny informace v jednom dokumentu. Tento přístup s sebou přináší několik problémů. Hlavní nevýhodou je fakt, že dokument nabývá

na objemu. Často je velikost jednoho dokumentu paměťově omezená, a proto je tento způsob nevhodný pro případy, ve kterých ukládáme větší množství dat. Výhodou je to, že proto abychom získali všechna potřebná data nám stačí pouze jeden dotaz nad jednou kolekcí. [17]

Obrázek 6: Reprezentace vztahu použitím objektů v poli

Způsoby uložení dat v Redis

Stejně tak jako dokumentové databáze, tak i databáze typu klíč-hodnota lze použít jako primární perzistentní úložiště pro jednoduché aplikace. V praxi se moc často nesetkáme s případy, kdy by byl celý systém postaveny pouze na tomto typu úložiště. Je to z důvodu velmi náročné údržby těchto databází. Ovšem stále častěji se stává, že jsou tyto databázové systémy použity jako primární úložiště pro určitý okruh dat. [18]

Problémem databází typu klíč-hodnota je transakční zpracování a ne-perzistentní povaha. Většina databází typu klíč-hodnota pracuje primárně s hlavní pamětí počítače (RAM). Redis a další podobné databázové systémy sice nabízí možnost perzistentního uložení na disk, ale nejedná se o synchronní operace (zapsaná data jsou převáděná na disk v určitých intervalech). I přesto je velkou výhodou těchto databází jejich rychlost a jednoduchost. Proto se hodí v případě, kdy požadujeme maximální propustnost i za cenu toho, že v případě výpadku přijdeme o určitý objem dat. Proto bychom před nasazením tohoto databázového systému měli zvážit, zda jsou pro nás tyto nedostatky přijatelné a opravdu potřebujeme maximální propustnost, kterou nám tyto databázové systémy nabízejí. V podstatě existují dva základní principy, jak v takových databázích uchovávat strukturovaná data.

První možností, která byla zmíněná již v kapitole o cache pamětech je využití identifikátoru daného záznamu jako klíče a ostatní data serializovat a následně je ukládat jako hodnotu v

podobě řetězce. Tento postup je jednoduchý a přímočarý. Nevýhodou je nutnost serializace a deserializace při čtení a zápisu. S tímto souvisí také fakt, že při každém dotazu, získáme všechna data, což by mohlo být nežádoucí v případě, že jsou data rozsáhlá a my pracujeme pouze s určitou části dat.

Další možnost je komplikovanější a způsobuje určité problémy při čtení dat. Princip spočívá v tom, že použijeme pro klíč hodnotu ve tvaru objekt:id:položka. Pokud bychom tedy chtěli uložit záznam o knize, náš klíč by mohl vypadat následovně kniha:1:nazev. Jako hodnotu bychom pak vložili konkrétní název knihy.

Problémem tohoto přístupu je v tom, že pro každou položku je nutné vytvořit jeden záznam v databázi. Při čtení je pak nutné provést tolik dotazů, kolik požadujeme položek na výstupu. Pokud bychom tedy chtěli získat Rok vydání, Název a informaci o nakladatelství jedné konkrétní knihy museli bychom provést 3 dotazy do databáze. Výhodou je pak možnost využití všech datových typů, které Redis nabízí.

2.6 Nestrukturovaná data v relačních databázích

Relační databáze nejsou ze své podstaty uzpůsobeny k tomu, aby uchovávala nestrukturovaná data. Ve většině informačních systémů jsou však situace, které nabádají k uložení dat v nestrukturované podobě. V relačních databázích se tento problém nejčastěji řeší tím, že se provede serializace požadovaných dat do některého z formátu se samo-popisným charakterem (JSON, XML) a uloží se jako řetězec. Můžeme tak ukládat strukturovaná data společně s nestrukturovanými.

Již několik let existuje pro výše zmíněné formáty SQL standard (SQL/XML, SQL/JSON), který ale v mnoha databázových systémech není implementován, anebo je implementován pouze částečně. Práce s nestrukturovanými daty je proto v různých relačních databázových systémech odlišná což vede k několika problémům, které jsou zmíněny níže.

Základní SQL/XML standard definuje datový typ pro práci s XML dokumenty, jejich mapování na tabulky, predikáty (`IS DOCUMENT`) a funkce (`XMLAGG`, `XMLCONCAT`, `XMLCOMMENT`, `XMLELEMENT`, `XMLFOREST`, ...). Později byly do tohoto standartu přidány možnosti přístupů k dokumentu pomocí dotazovacího jazyka XQuery (jazyk, primárně navržený pro dotazování nad daty uloženými ve formátu XML). [19]

Pro standard SQL/JSON je překvapující, že nedefinuje nativní datový typ tak jak je to u standardu pro XML. Namísto toho je ve standardu definováno uložení dat v JSON formátu jako řetězec. Z tohoto důvodu je požadováno, aby všechny funkce pracující s daty v JSON formátu byly schopný parsovat data z řetězce. I přesto některé databázové systémy poskytují pro JSON nativní datový typ. Jedinou nevýhodou, která plyne z použití řetězce pro uložení JSON je nemožnost automatické kontroly validity formátu (je nutné použít „CHECK“ omezení). Kromě toho

definuje standard podobu použitého formátu a několik funkcí sloužící pro vytváření a průchod JSON daty. [20]

V následujících odstavcích jsou popsány možnosti práce s JSON a XML formáty v nejznámějších a nejrozšířenějších implementacích relační databáze.

Microsoft SQL Server

Pro ukládání dat v XML formátu, nám SQL Server nabízí nativní datový typ (podle standardu). Využití tohoto datového typu není pro uložení XML povinné, ale přináší nám několik výhod. Mezi největší výhody, které plynou z jeho použití patří automatická kontrola validity dat a rychlost parsování. Tento způsob uložení dat je nejvýhodnější, pokud požadujeme následující operace:

- Požadujeme přímý způsob, jak ukládat XML data na serveru a současně zachovat pořadí dokumentu a strukturu dokumentu
- Pokud není vyžadováno schéma XML dokumentu
- Požadujeme procházení a případně i úpravu XML dokumentu
- Požadujeme indexování dokumentu pro rychlejší procházení

Další možnosti, jak uložit XML do SQL databáze je využití datových typů `nvarchar` nebo `varbinary`. Tento přístup je výhodný převážně tehdy, pokud požadujeme přesnou textovou kopii celého XML dokumentu a nevyžadujeme, aby měl XML dokument specifikované schéma. Pokud ovšem potřebujeme často procházet přes XML dokument je tento přístup z hlediska výkonnosti neefektivní. V praxi se pak pro případy, kdy chceme využít výhod plynoucí z uložení dat v nativním formátu a zároveň požadujeme možnost získání textové kopie, využívá kombinace obou technik, kdy pro procházení dat použijeme nativní datový typ a pro získání textové podoby dokumentu jsou data uložena v jednom z výše zmíněných formátů.

Kromě uložení dat ve formě řetězce nebo v odpovídající datovém typu je možné provést mapování XML dokumentu na sloupce v jedné nebo více tabulek. Tento přístup je taktéž součástí standardu a vyžaduje, aby měl XML dokument anotované schéma (AXSD), pomocí kterého se budou rozdělovat data do příslušných sloupců a tabulek. Nevýhodou je to, že schéma nemůže obsahovat rekurzivní záznamy.

SQL Server poskytuje (podle standardu) mnoho vestavěných metod a funkcí pro práci s XML dokumentem se širokou podporu pro dotazovací jazyk XQuery, pomocí kterého lze jednoduše filtrovat výstupní data podle obsahu XML dokumentu. Následuje seznam funkcí používaných při práci s XML formátem. [21] [22]

- **query** - vrací XML hodnotu na základě specifikované cesty

- **value** - vrací skalární hodnotu získanou z XML dokumentu na základě specifikované cesty
- **exists** - zjišťuje, zda výstup na základě specifikované cesty není nulový
- **modify** - funkce umožňující změnu XML dokumentu

Kromě těchto základních funkcí, nabízí MSSQL predikát `FOR XML`, který převede výstup z SQL dotazu do podoby XML dokumentu. Jedná se o proprietární nástroj specifický pro tento databázový systém poskytující implementaci standardu SQL/XML pro mapování SQL na XML.

Podpora pro standard SQL/JSON není v SQL Serveru tolik rozšířená jako podporovaná standardu SQL/XML. Je definováno pouze několik základních funkcí, které usnadňují práci s daty v JSON. V následujícím seznamu jsou vypsány nejpoužívanější funkce:

- **ISJSON** - kontroluje, zda je vstupní řetězec validní JSON řetězce
- **JSON_VALUE** - extrahuje skalární hodnoty z JSON řetězce
- **JSON_QUERY** - extrahuje objekt nebo pole z JSON řetězce
- **JSON_MODIFY** - mění hodnoty v JSON řetězce

Pomocí těchto funkcí lze jednoduše a efektivně manipulovat, filtrovat a upravovat data ve formátu JSON. Všechny tyto funkce používají pro odkazování na objekty uvnitř JSON řetězce syntaxi založenou na programovacím jazyce Javascript (také definováno ve standardu). [23]

Oracle SQL Server

Oracle SQL Server zachází při implementaci XML formátu ještě o kus dál než SQL Server. Kromě toho, že široce podporuje standard SQL/XML, nabízí také možnost využití nativního XML databázového uložiště, který je distribuován současně ve všech verzí Oracle databáze (od verze Oracle 9iR2). Oracle XML DB poskytuje pro všechny klíčové XML standardy jako jsou jmenné prostory (namesapce), DOM, XQuery nebo XSLT. Vývojáři jsou proto schopni využívat techniky zaměřené na XML pro ukládání, správu, organizaci a manipulaci s XML obsahem uloženým v databázi. [24]

Oracle databáze poskytuje nativní datový typ (XMLType), který představuje abstrakci poskytující několik modelů pro uložení dat. Od verze Oracle Database 12c Release 2 jsou podporovány následující tři hlavní modely pro uložení dat:

- **Binární XML uložiště** - ukládá XML pomocí nativní binární reprezentace dat
- **Objektově relační XML uložiště** - ukládá XML dokument jako množinu SQL objektů
- **Relační uložiště** - s použitím SQL/XML a XQuery je možné vytvořit pohledy, které reprezentují relační data jako hodnoty v XMLType

Pro práci s XML samotným nabízí Oracle sadu vestavěných funkcí. Všechny z těchto funkcí využívají pro svou práci jazyky XQuery a XPath. V následujícím seznamu jsou stručně popsány nejpoužívanější funkce používané při manipulaci s daty uloženými ve formátu XML.

- **XMLQuery** - vrací obsah XML dokumentu na základě stanovené cesty
- **XMLTable** - mapuje výsledek dotazu v XML dokumentu na relační řádky a sloupce
- **XMLExists** - kontroluje, jestli zadaná cesta v XML dokumentu nevrací prázdnou hodnotu
- **XMLCast** - slouží pro přetypování skalární hodnoty v XML dokumentu na datový typ použitelný v SQL

Kromě těchto standardních funkcí poskytované na základě standardu, Oracle nabízí také možnost práce s XML na úrovni DOM. K tomu slouží dva balíčky, které jsou součástí procedurální nadstavby PL/SQL s názvem `DBMS_XMLDOM` a `DBMS_XMLPARSER`.

Oracle ve svém databázovém řešení s podporou standardu SQL/JSON také nezaostává a poskytuje implementaci pro širokou škálu definovaných funkcí. K těmto funkcím je přidáno ještě několik funkcí specifických pouze pro tento databázový systém. Jsou zde tedy přítomné základní dotazovací funkce, které byly zmíněny již u SQL Servery, jako jsou `JSON_VALUE`, `JSON_QUERY` nebo `IS_JSON`. Kromě dotazovacích funkcí Oracle také poskytuje implementaci pro funkce SQL/JSON standardu sloužící pro generování JSON z SQL dat. Mezi tyto funkce patří `JSON_OBJECT` nebo `JSON_ARRAY`. [25]

Stejně tak jako u SQL Serveru, Oracle nenabízí žádný nativní datový typ pro ukládání JSON dat. Tyto data se tedy typicky ukládají do sloupců s datovými typy `VARCHAR2`, `CLOB` nebo `BLOB`. V procedurální nadstavbě PL/SQL jsou přítomny objektové typy pro usnadnění práce při manipulaci s JSON daty. Pomocí těchto typů lze jednoduše prozkoumávat, měnit nebo serializovat (zpátky do textové podoby) JSON data.

PostgreSQL Server

PostgreSQL za výše zmíněnými databázovými systémy nezaostává a také poskytuje podporu standardu SQL/XML a s ním nativní datový typ pro práci s XML a mnoho funkcí pro usnadnění práce. Implementace je vytvořena téměř pro všechny standardem definované funkce a predikáty. Následující seznam shrnuje nejpoužívanější funkce pro práci s XML podle standardu SQL/XML tak jak je implementuje PostgreSQL. [26]

- **Xmlcomment** - vytváří XML hodnotu obsahující XML komentář se specifikovaným textem a obsahem
- **Xmlconcat** - funkce spojující list jednotlivých XML hodnot
- **Xmlelement** - vytváří XML element s požadovaným názvem, atributy a obsahem

- **Xmlforest** - vytváří XML strom ze zadaných elementů
- **Xmlroot** - mění vlastnosti kořenového uzlu v XML dokumentu
- **Xmlagg** - je podobná jako Xmlconcat, spojuje XML hodnoty na základě specifikovaných sloupců

Další definicí standardu SQL/XML je mapování výstupu z SQL dotazu na XML. K tomuto účelu PostgreSQL nabízí také několik funkcí. K těmto funkcím přibývá ještě rozšíření, které poskytuje funkce jako `xpath_exists`, `xml_validate` a další predikáty pro kontrolu formátu jako `xml_is_well_formed`. [27]

Na rozdíl od obou předchozích databázových systémů existuje v PostgreSQL nativní datový typ json pro ukládání dat (od verze 9.2). V pozdějších verzích byl také přidán další datový typ `jsonb`. Rozdíl mezi těmito datovými typy je v tom, že `json` ukládá data ve formě klasického řetězce, zatímco `jsonb` používá binární formát, který způsobuje to, že bude (kvůli nutnosti konverze) částečně zpomaleno vkládání dat, ale zpracování takto uložených dat bude podstatně rychlejší. `Jsonb` také umožňuje využití indexů. Oba výše zmíněné typy zaručují, že data v těchto sloupcích budou validní podle pravidel JSON formátu. [28]

Stejně jako Oracle, tak i PostgreSQL nabízí několik užitečných funkcí pro práci s JSON dokumentem a přidává k nim i speciální notaci (podobné jako v MySQL), pomocí které lze jednoduše procházet JSON dokument. Následuje výčet nejpoužívanějších funkcí pro práci s JSON dokumentem (tento výčet je pouze pro ukázku možností práce s JSON a neobsahuje všechny funkce, které PostgreSQL nabízí).

- **to_json** - převádí jakýkoliv element do formátu JSON (pro datový typ jsonb existuje obdobná funkce to_jsonb)
- **array_to_json** - převádí pole do JSON
- **row_to_json** - jako vstup je řádek tabulky, který je převeden do JSON formátu
- **json_object** - sestavuje JSON objekt z textového pole
- **json_each** - rozvádí JSON objekt na množinu klíč-hodnota
- **jsonb_pretty** - vrací JSON s odpovídajícím odsazením a řádkováním

Mnoho funkcí implementovaných v PostgreSQL není definováno ve standardu a je specifických pouze pro tento databázový systém.

MySQL Server

Na rozdíl od všech předchozích databázových systémů, MySQL nabízí pouze základní implementaci standardu SQL/XML. Neexistuje zde žádný nativní datový typ pro uložení dat seriali-

zovaných do tohoto formátu. Namísto toho je nutné data ukládat jako řetězec do datových typů `CLOB`, `TEXT` nebo podobných.

I přesto, že zde není veliká podpora pro tento formát, tak MySQL nabízí pár základních funkcí, které usnadňují manipulaci s uloženými daty. Všechny tyto funkce podporují dotazovací jazyk XQuery (podle standardu) pro orientaci v XML dokumentu. Mezi dvě nejpoužívanější funkce patří `ExtractValue` a `UpdateXML`. Funkce `ExtractValue` slouží pro získání skalární hodnoty z XML dokumentu podle zadané cesty. Funkci `UpdateXML` lze použít pro úpravu XML dokumentu. Pomocí této funkce tak lze přidávat nové uzly, mazat již existující uzly nebo měnit jejich hodnoty. [29]

Podpora standardu SQL/JSON je v MySQL rozšířená daleko více. Existuje zde nativní datový typ (není součástí standardu). Při použití tohoto datového typu budou JSON dokumenty ukládány ve speciálním vnitřním formátu, který umožňuje rychlou manipulaci s daty. Vnitřní binární formát JSON je strukturován tak, že umožňuje serveru vyhledávat hodnoty v JSON dokumentu na základě jednoho nebo více klíčů, což je velice efektivní. Důležitým omezením pro tento datový typ je nemožnost nastavení výchozí hodnoty a přímé vytvoření indexu nad tímto sloupcem. Indexy se vytvářejí pomocí virtuálních sloupců. Koncept virtuálních sloupců v MySQL existuje od verze 5.7. Virtuální sloupce se od klasických sloupců liší tím, že nejsou perzistentně uloženy v databázi. Namísto toho se generují dynamicky pro každý záznam v tabulce. Tyto sloupce lze tedy použít například pro slučování hodnot několika sloupců nebo pravě pro vytvoření sloupců, které budou extrahovat hodnoty z JSON dokumentu na základě specifikované cesty. Nad těmito sloupci lze pak vytvářet indexy, které umožňují rychlejší vyhledávání záznamů v tabulce. [30] [31]

Pro vyhledávání a filtrování v JSON dokumentu MySQL nabízí tzv. operátor sloupcové cesty, který rozšiřuje možnosti dotazovacího jazyka pro JSON definovaného ve standardu a umožňuje jednodušší a přehlednější zápis. Pomoci něj lze jednoduše vybírat požadované části dokumentu. Pro ostatní manipulaci s JSON dokumentem jsou podle standardu přítomny vestavěné funkce.

V kapitole 3.3 jsou uvedeny ukázky funkcí a způsobů, jakým MySQL manipuluje s dokumenty uloženými ve formátech JSON a XML.

Hlavní výhodu, kterou přináší možnost uložení nestrukturovaných dat ve formátu JSON nebo XML je přiblížení relačních databází ke schopnostem NoSQL databází, a to převážně z hlediska ukládání nestrukturovaných dat, které jsou v poslední době stále častější. Další nespornou výhodou je možnost uložení nestrukturovaných dat společně s daty relačními, což je v mnoha ohledech výhodné a často využívané.

Některé frameworky (především ORM frameworky) však nejsou na práci s takto uloženými daty ještě připraveny, a proto se v mnoha případech stává, že při použití těchto frameworků nám

nezbude nic jiného než pro manipulaci s daty uloženými v jednom z výše zmíněných formátů použít parsovací nástroje, které nabízí programovací jazyk, se kterým pracujeme. Velké množství odlišných názvu funkcí a fakt, že ne každý relační databázový systém nabízí tutéž funkcionalitu komplikuje vývojářům přechody mezi různými databázovými systémy.

2.7 NoSQL databáze v prostředí Cloudu

Využívání „server-less“ architektury [32] a s tím souvisejícího cloudového řešení pro vývoj aplikací, se čím dá více rozvíjí. Cloudy poskytují poměrně snadné řešení, jak se vyhnout zdlouhavému návrhu a nastavování fyzické infrastruktury pro vyvíjenou aplikaci. Díky tomu se mohou vývojáři soustředit na vývoj samotné aplikace a nezatěžovat se správou fyzické infrastruktury.

Hlavní výhody vyplývající z využití cloudového řešení je snížení finančních nákladů (často nám poskytovatelé cloudu nabízejí možnosti platit pouze za ty služby, které opravdu využíváme), jednoduchá škálovatelnost (navýšení, popřípadě snížení výpočetní kapacity je velice jednoduché), bezpečnost (data na cloudu bývají velmi často automaticky zálohována, aby nedošlo k jejich ztrátě) a přístup k automatickým aktualizacím (software běžící v cloudu je většinou automaticky aktualizován bez zásahu uživatele).

Existuje mnoho cloudových řešení. Každé z nich nabízí jinou úroveň poskytovaných služeb a cenu. Mezi nejznámější cloudové řešení patří **Azure** od společnosti Microsoft, **Google Cloud Platform** a **AWS** (Amazon Web Service). Každá z těchto platforem nabízí možnost bezplatného nasazení některé NoSQL databáze. Například v případě AWS lze využít bezplatné nabídky pro vytvoření vlastní instance databázového systému Amazon DynamoDB s velikostí úložiště omezené na 25GB. Samozřejmě tyto bezplatné služby mají určitá omezení a pro nasazení reálné aplikace je výhodnější využít placené nadstavby, které nabízejí daleko propracovanější možnosti konfigurace.

Ve většině případů je pro konfiguraci určité služby v cloudu musíte přihlásit do webového rozhraní daného řešení a vybrat si službu, o kterou máte zájem. MongoDB nabízí vlastní nástroj **MongoDB Atlas** pro vytvoření a správu databáze MongoDB v cloudu. Poskytuje jednoduché webové rozhraní, pomocí kterého lze za pár minut vytvořit plně funkční instanci databázového systémy na jedné ze tří výše zmíněných platforem. Celý proces vytvoření databáze na platformě Azure je představen v kapitole 3.4. [33]

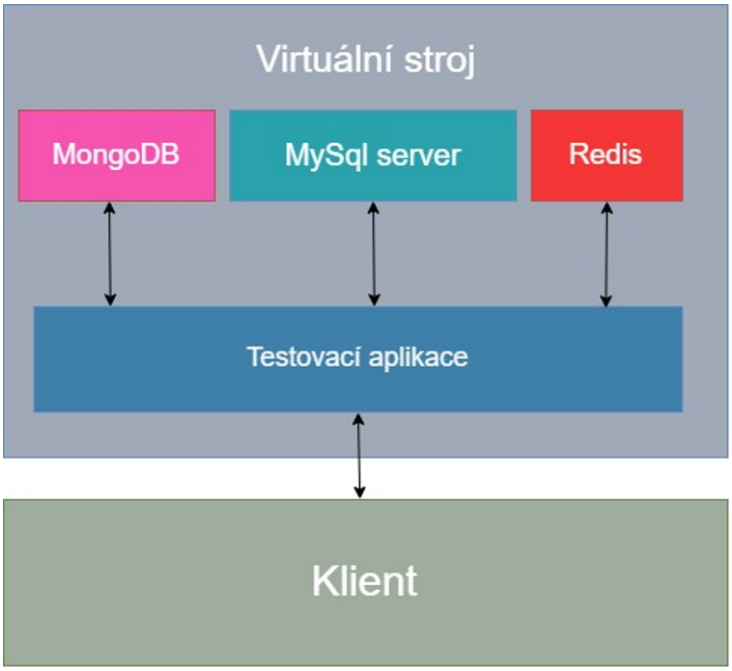
3 Praktická část

Začátek praktické části je zaměřen na otestování výkonu databázových systémů a jejich porovnání. Bude zde popsána architektura testovací aplikace její konfigurace a hardwarové specifikace. Další část se věnuje základním způsobům práce s NoSQL databázemi v programovém prostředí. Zde bude největší část věnována databázovému systému MongoDB. Následně budou provedeny ukázky možností práce s nestrukturovanými daty v prostředí relačních databází, a to konkrétně v databázovém systému MySQL. Poslední část obsahuje ukázku toho, jak lze využít systém MongoDB Altas pro nasazení databázového systému MongoDB na cloudovou platformu Azure.

3.1 Porovnání výkonu NoSQL a relačních databází

Způsob testování byl zvolen na základě snahy se co nejvíce přiblížit reálným situacím. Hlavním cílem této kapitoly není popsání teoretických rychlostí v laboratorním prostředí, ale zhodnocení výkonů vybraných databází při reálném způsobu dotazování, a to se zaměřením na velké vytížení aplikace.

3.1.1 Architektura a technologie použité v průběhu testování



Obrázek 7: Architektura testování

Základní popis způsobu testování

Pro testování výkonu jednotlivých případu nasazení je vytvořena webová aplikace (Web API). Testovací aplikace na základě typu požadavku získává data z databázových serveru, které jsou

spuštěny na stejném virtuálním stroji jako je samotná testovací aplikace. Z aplikace jsou vystaveny přístupové body pro rozhraní API. Pro získání maximální propustnosti jsou všechny přístupové body implementovány pomocí asynchronních metod[3] a využívají asynchronních implementací metod, nabízených v NuGet balíčních **MongoDB.Driver** a **MySql.Data**.

Samotné testování probíhá tak, že je na určitém počtu zařízení spuštěna klientská aplikace. Ta v určitých intervalech zasílá předem specifikovaný počet dotazů do testovací aplikace. Dotaz je v této aplikaci zpracován a získaná data včetně časových údajů jsou zaslána zpět do klientské aplikace, která je následně ukládá. Samotná data jsou irelevantní a jsou posílány pouze za účelem změření času potřebného pro jejich přenos sítí.

Webové API standardně využívá JSON pro zformátování výstupu. Všechny přístupové body, jež budou provádět testování budou vracet JSON řetězec s formátem popsaným v tabulce 3.

Klíč	Hodnota
QueryTimeElapsed	Časový údaj vyjadřující dobu vykonávání dotazu v databázi
ConvertTimeElapsed	Nepovinný časový údaj vyjadřující dobu potřebnou pro převedení surových dat do formátu JSON
TotalTimeElapsed	Časový údaj vyjadřující sumarizaci časových údajů z `QueryTimeElapsed` a `ConvertTimeElapsed`
Data	Samotná data v JSON formátu

Tabulka 3: Formát návratové zprávy testovací metody

V následujícím seznamu jsou popsány technologie použité v průběhu testování:

- **MySQL**[4] (verze 5.7) - relační databázový systém
- **Redis** (verze 3.0.6) - databázový systém typu klíč–hodnota
- **MongoDB** (verze 2.6.10) - dokumentová databáze
- **.NET Core** 3.0 - multiplatformní framework pro vývoj (webových) aplikací
- **Nginx** (verze 1.10.3) - webový server
- **Kestrel** - integrovaný webový server do platformy .NET Core
- **Ubuntu** (verze 16.04) - linuxová distribuce
- **Windows 10** - operační systém

[3] Asynchronní metody neblokují vlákno, ve kterém jsou spuštěny. Je tedy možné vykonávat několik dotazů souběžně.

[4] Pro testovací tabulky je použito InnoDB.

Serverová část testovacího systému je spuštěna na virtuálním stroji jehož parametry jsou popsány v tabulce 4. V této tabulce lze také nalézt hardwarové specifikace pro zařízení, na kterých byla spouštěna klientská aplikace.

	Server	Client
Frekvence procesoru [Ghz]	2,4	3,9
Počet jader	8	6
Paměť RAM [GB]	8	16
Typ disku	SSD	HDD
Velikost disku [GB]	250	–

Tabulka 4: Hardwarová specifikace testovacího systému

Popis nastavení a nasazení testovacího webového serveru
Platforma .NET Core je nástupcem starší platformy .NET Framework. Doménou této platformy je možnost multiplatformního vývoje aplikací. Díky tomu je možné spustit aplikace naprogramované v C#, F# nebo VisualBasic i na platformě Linux a MacOS, což v předešlé verzi nebylo možné.

Součásti platformy .NET Core je i webový server Kestrel [34]. Tato technologie je součásti frameworku proto, aby se zachovala možnost multiplatformního vývoje. Při nasazování aplikace je pak primární webový server (IIS, Apache, Nginx) použit jako proxy server, který směřuje všechny požadavky na webový server Kestrel. Ten pak kontroluje přístup k samotné aplikaci. Hlavním cílem tohoto řešení je jednotná konfigurace napříč různých platforem.

Proto abychom byli schopni spustit .NET Core aplikaci na Linux platformě, je nutné ji nainstalovat. Následující příkazy nainstalují běhové prostředí pro tuto platformu na operačním systému Ubuntu.

```
wget -q https://packages.microsoft.com/config/ubuntu/16.04/packages-microsoft-
    prod.deb -O packages-microsoft-prod.deb

sudo dpkg -i packages-microsoft-prod.deb

sudo apt-get update
sudo apt-get install apt-transport-https
sudo apt-get install dotnet-sdk-3.0
```

Výpis 1: Příkazy pro instalaci běhového prostředí .NET na plataformě Linux

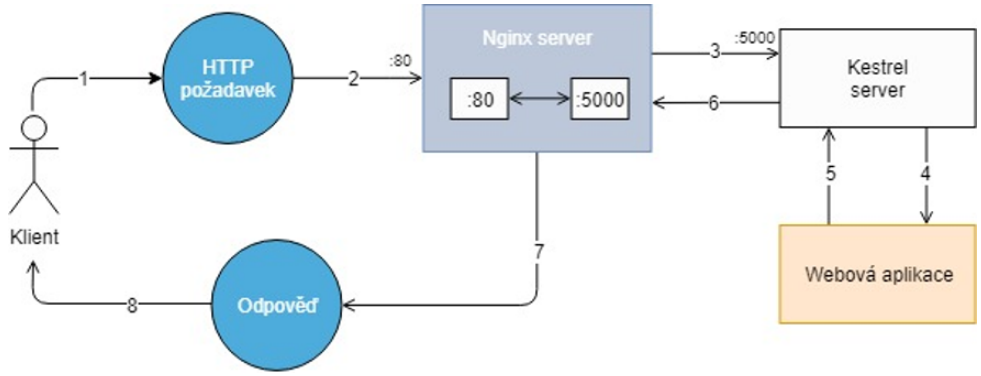
Jakmile jsou tyto kroky provedeny, je možné spouštět aplikace vytvořené v platformě .NET Core. Dalším nutným krokem k tomu, abychom byli schopni přistupovat k naší webové aplikaci z prostředí internetu, je konfigurace primárního webového serveru tak, aby pracoval jako proxy server a směroval komunikaci na webový server Kestrel.

Následující ukázka konfigurace nastaví webový server Nginx tak, aby fungoval jako webový proxy server a přesměrovával veškeré požadavky, přijaté na portu 80, na port 5000. Na tomto portu ve výchozím stavu naslouchá webový sever Kestrel (pomoví konfigurace lze změnit), pod kterým běží naše testovací webová aplikace.

```
location / {
proxy_pass http://localhost:5000;
proxy_http_version 1.1;
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection keep-alive;
proxy_set_header Host $host;
proxy_cache_bypass $http_upgrade;
}
```

Výpis 2: Konfigurační soubor pro nastavení Nginx jako proxy server

Na obrázku 8 je znázorněn proces zpracování požadavku odeslaného do testovací aplikace. Popis celého procesu je pro přehlednost velmi zjednodušen. Ve skutečnosti mezi vrstvou serveru Kestrel a samotnou aplikací probíhá mnoho dalších procesů, které však nejsou pro tuto práci důležité a z tohoto důvodu zde nejsou zmíněny.



Obrázek 8: Zpracování HTTP požadavku

3.1.2 Testovací scénáře a jejich výsledky

Ve všech testovacích scénářích bylo na testovací server souběžně odesláno přesně 200 dotazů z každého testovacího klienta. Počet testovacích klientů, kteří odesílali požadavky bylo 15. Celkové

množství dotazů v každém jednotlivém testovacím scénáři bylo 3000. Každá odpověď na dotaz obsahovala informace popsané v tabulce 3 nezávisle na ostatních klientech. Časy získané ze všech měření byly zprůměrovány a výsledky převedeny do následujících tabulek a grafů.

V tabulce 5 jsou znázorněny výsledky z první sady testování. Testování mělo za cíl zjistit, jaký vliv má hodnota TTL na výkon při dotazování velkého objemu dat. Předpoklad byl takový, že při změně hodnoty TTL by měl být čas potřebný pro vykonání dotazu tím nižší, čím vyšší by byla hodnota TTL. Mělo by to být důsledkem toho, že pro větší hodnoty TTL dochází k častějšímu získání hodnot z cache pamětí (cache hit) a není tak potřeba zasahovat do primární paměti, kterou byla v tomto případě relační databáze MySQL. Pro reprezentaci cache paměti zde byl použit NoSQL databázový systém Redis, který tvořil mezivrstvu mezi primární databází a testovací aplikaci. Testovací aplikace pro tuto sadu testů byla implementována tak, aby využívala cachovací strategie Lazy load.

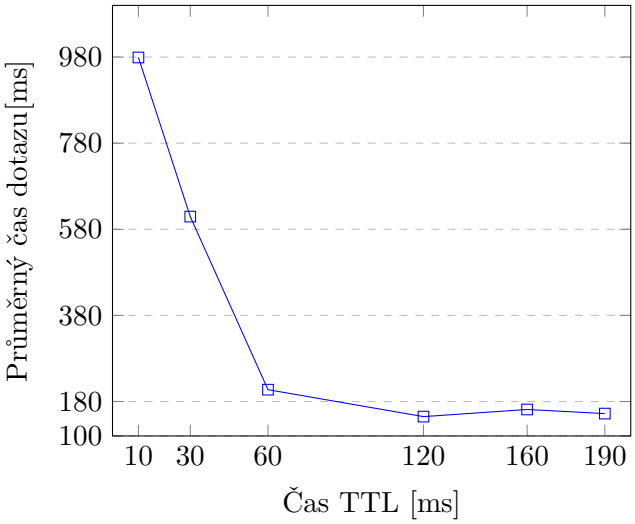
Dotazy vykonávané v první testovací sadě byly na bázi full-textového vyhledávání. Výběr tohoto typu dotazů byl motivován jejich náročností a dlouhému vykonávání dotazu.

	Čas TTL [ms]					
	10	30	60	120	160	190
Čas výkonání dotazu [ms]	979,04	609,55	207,50	144,95	161,53	151,96
Čas klienta [ms]	1008,58	635,67	237,67	170,49	190,12	182,82
Čas přenosu síti [ms]	29,54	26,12	30,17	25,54	28,59	30,85
Čas převodu do JSON [ms]	0,66	0,30	0,16	0,09	0,09	0,09
Cache Hit [%]	84,03	90,33	95,97	97,27	97,07	97,20
Cache Miss [%]	15,97	9,67	4,03	2,73	2,93	2,80

Tabulka 5: Výsledky měření s použitím cache paměti

Jak můžeme vidět v tabulce 5, tak čas potřebný pro převod dat do formátu JSON se pohybuje řádech stovek mikrosekund a je tedy v podstatě zanedbatelný. Čas potřebný pro přenos dat sítí se zdá již podstatně větší, ale v porovnání s časem vykonávání samotného dotazu v databázi je tento časový údaj stále zanedbatelný. Nejpodstatnějším místem kde vzniká prodleva je tedy samotný databázový systém a vykonávání samotného dotazu.

Jedním z hlavních předpokladů bylo, že se bude čas potřebný pro vykonání dotazu podstatně snižovat s navyšováním hodnoty TTL. Tento předpoklad se prokázal pro hodnoty mezi 10 a 120 sekundy. Mezi těmito hodnotami došlo k průměrnému snížení časů vykonání jednoho dotazu přibližně o 834 milisekund a procentuálního navýšení hodnoty "cache hit"o 13%. Naproti tomu pro vyšší hodnoty TTL se časy téměř neliší.



Obrázek 9: Graf závislosti času vykonání dotazu na hodnotě TTL

Základní shrnutí tabulky 5 lze vidět v grafu 9. Zde je také dobře vidět rozdíl změny času potřebného pro vykonání dotazu mezi nízkými a vysokými hodnotách TTL.

Při porovnávání výsledku hraje roli také časový průběh samotného testu. Pokud by test probíhal po delší dobu, pak by mezi těmito hodnotami mohlo docházet k větším odchylkám.

Další sada testů probíhala mezi databázovými systémy MongoDB a MySQL. Podstatou této sady testů bylo zjistit, o kolik bude relační databáze pomalejší oproti NoSQL databází (v tomto případě MongoDB). Z tohoto důvodu byl pro databázi v MySQL použitý engine InnoDB, který podporuje transakční zpracování a z tohoto důvodu mnohem lépe odpovídá obecnému pojetí relačních databází. Důvodem, proč jsou relační databáze obecně považovány za méně výkonné je právě toto transakční zpracovávání dat Předpokladem tedy bylo, že dotazy na databázový systém MongoDB budou zpracovány daleko rychleji, než tomu bude v případě MySQL serveru.

	MongoDB	MySQL (vnitřní cache)	MySQL
Čas výkonání dotazu [ms]	1433,91	5499,03	6523,29
Čas klienta [ms]	1464,37	5578,91	6677,01
Čas přenosu síti [ms]	30,47	79,88	153,71

Tabulka 6: Výsledky měření mezi MongoDB a MySQL

V tabulce 6 lze vidět průměrné časy vykonávání dotazů pro databázový systém MongoDB a MySQL. Je důležité si všimnout, že tabulka obsahuje dva sloupce pro MySQL. Je to z toho důvod, že samotný MySQL server poskytuje vlastní implementaci vnitřní cache paměti. Třetí

sloupec obsahuje časové údaje pro variantu se zapnutou vnitřní cache paměti, zatímco ve čtvrtém sloupci jsou uvedeny časové údaje pro MySQL server s vypnutou vnitřní cache paměti. Výsledky jsou získávány ze stejné kolekce dotazů jako jsou použity v tabulce 5 bez použití jakékoliv externí cache paměti.

Výsledky výkonnostního testu víceméně odpovídají předpokladům. Je zde vidět, že časy přenosu data sítí jsou zde velice rozlišné a v jednom případě překonali hranici 100 milisekund, nicméně z hlediska celkového času vykonávání dotazu klienta téměř nehrají roli. Dále lze vidět, že průměrný čas vykonávání samotného dotazu v MongoDB se oproti MySQL snížil o více než 4000 milisekund a v případě vypnutého vnitřního cachovacího procesu v MySQL dokonce o více než 5000 milisekund. Tyto rozdíly jsou již velice značné a je tedy vidět, že transakční zpracovávání velice ovlivňuje výkon databázového serveru.

Tyto výsledky je také možné porovnat s výsledky z tabulky 5. Jak si lze všimnout, tak již pro nejnižší testovací hodnotu parametru TTL (10 s) se časy oproti samotnému MySQL serveru snížili velice podstatně. Roli zde nehraje ani to, zda je vnitřní cache MySQL serveru vypnutá či zapnuta. I přesto, že lze pomocí konfigurace paměťový prostor pro vnitřní cache paměť upravit, je stále tato implementace cache paměti velice omezená ve svých schopnostech a lze vidět, že se nemůže rovnat NoSQL databázi.

Při dalším porovnání zjistíme, že i při použití databázového systému MongoDB jsou časy vykonávání dotazu vyšší než pro MySQL server s cachovací mezivrstvou ve formě Redis databáze. Pro nízké hodnoty TTL není rozdíl v čase až tak výrazný, nicméně se zvyšující se hodnotou TTL rozdíl podstatně narůstá.

Toto zjištění se může zdát irelevantní. Ale již v teoretické části bylo zmíněno, že prvním krokem pro zlepšení výkonu nasazených relačních databází je přidání vrstvy mezi aplikaci a databázi, která slouží jako vzdálena cache paměť. Zde lze na praktickém příkladě vidět, že přidání této mezivrstvy je velice účinné z hlediska výkonu, který je větší než pro databázi nativně vytvořenou v NoSQL technologii. Hlavním problémem tedy zůstává zachování konzistence mezi primární databází a cache paměti.

Všechny výše zmíněné testovací scénáře pracovaly s full-textovými dotazy. Tyto dotazy jsou obecně velmi náročné a z tohoto důvodu byly časy vykonávání dotazů velice vysoké. Velice často se ovšem sestavují dotazy, které provádějí operace nad indexovanými atributy. Pokud atribut obsahuje index, není databázový systém nucen procházet celou databázi. Místo toho prochází speciálně předpřipravenou datovou strukturu (nejčasteji nějakou formu vyvažujícího se stromu). Efektivita dotazů, jež filtrují na základě indexovaných atributů je daleko vyšší než v případě full-textového vyhledávání. Samozřejmě lze také vytvořit full-textové indexy. V předchozích scénářích však nebyly použity žádné full-textové indexy z toho důvodu, že je uložený systém InnoDB nepodporuje.

Následující testovací scénář porovnává výkon MySQL a MongoDB, jež provádějí dotazy pouze nad indexovanými atributy. Předpokladem pro tuto sadu testů stále zůstává, že by měl být databázový systém MongoDB výkonnější (stále počítáme s transakčním zpracováváním).

	MySQL	MongoDB
Čas výkonání dotazu [ms]	5,47	4,91
Čas klienta [ms]	20,56	33,92
Čas přenosu síti [ms]	13,10	29,01

Tabulka 7: Výsledky měření mezi MongoDB a MySQL (pouze indexované atributy)

V tabulce 7 lze vidět výsledky vykonaných testu pouze nad indexovanými atributy. Lze si všimnout, že se časy oproti předchozím měřením razantně snížily. Velkou roli zde začalo hrát i zpoždění při přenosu sítí, které bylo v předchozích případech téměř zanedbatelné. I přesto, že rychlost vykonání dotazu je v případě MongoDB větší, tak data se ke klientovi dostaly v průměru o přibližně 13 milisekund později.

Můžeme zde také vidět, že pokud jsou dotazy vykonávané nad indexovanými atributy, pak jsou rozdíly ve výkonu mezi MongoDB a MySQL téměř zanedbatelné a v tomto případě se liší o necelou 1 milisekundu.

Všechny výše zmíněné scénáře byly postaveny na získávání dat z databáze. V mnoha případech je však také velmi podstatná rychlost při vkládání záznamů, a to především u systémů, které získávají data ze zařízení IoT. Jedním z příkladů zařízení, jež potřebuje velmi vysokou propustnost při vkládání je například samořídící automobil, který je schopen za účelem analýzy provozu vložit do databáze velmi vysoké množství různých dat během několika minut.

Následující scénář srovnával rychlost vkládání dat mezi databázovými systémy MongoDB a MySQL. Způsob testování probíhal totožně jako v předchozích příkladech. Rozdílným je pouze formát návratové zprávy, který obsahuje pouze časový údaj reprezentující samotné vykonání dotazu. Předpokladem je opět to, že bude MongoDB oproti MySQL rychlejší. Předpoklad je postaven na faktu, že MongoDB využívá pro řízení souběhu optimistický přístup a MySQL s databázovým enginem pesimistický přístup.

	MySQL	MongoDB
Čas výkonání dotazu [ms]	1,05	25.77

Tabulka 8: Výsledky měření pro operaci Insert mezi MongoDB a MySQL

Výsledek testů tento předpoklad potvrdil. Průměrná hodnota času potřebného pro vložení jednoho záznamů do databázového systému MongoDB je 1.05ms, zatímco pro MySQL je to 25.77ms. Z výsledků je vidět podstatný rozdíl mezi oběma databázovými systémy. Po zaokrouhlení lze říct, že je MongoDB při vykonávání operace INSERT téměř 26x rychlejší. Toto je již pro některé systémy velmi podstatné číslo, které může rozhodovat při výběru databázové technologie pro daný systém.

Sumarizace výsledků testů

Z provedených testů lze vidět, že jsou NoSQL databáze oproti relačním databázím daleko výkonnější, a to jak v případě primární databáze, tak i jako podpůrné sekundární systémy. Výkonnostní rozdíl se ztrácí v případě, že jsou dotazovaná data indexována. Udržování indexů je však velice náročné v případě vkládání nových záznamů a z toho důvodu se musí přidělování indexů jednotlivým atributům velice důkladně zvážit.

Testy také ukázaly, že použití databázového systému Redis jako cachovacího serveru pro relační databázi je velice účinný krok, ale pouze v případě, že se smíříme s částečnou nekonzistencí dat. Pro časově náročné dotazy se v tomto případě dostáváme na nižší časové hodnoty, než při použití pouze samostatně stojící NoSQL technologie jako byl v případě testování databázový systém MongoDB, a to již při velmi nízkých hodnotách TTL. Na druhou stranu se z testů ukázalo, že navyšování hodnoty TTL je od určité hranice téměř nepodstatné a pouze prodlužuje negativní vliv tohoto přístupu, čímž je nekonzistence dat.

V případě srovnání rychlostí pro vkládání nových záznamů se rozdíl mezi systémy MongoDB a MySQL ještě více prohloubil. Ukázalo se, že je MongoDB při vkládání daleko výkonnější, a to téměř 26x.

3.2 Práce s NoSQL v klientském prostředí programovacích jazyků

Klientská i serverová část testovacího systému je vytvořena pomocí technologie .NET Core a programovacího jazyka C#. Tato technologie byla vybrána z hlediska jednoduché implementace asynchronních metod zvyšující rychlost při zpracování více souběžných požadavku, možnosti multiplatformního vývoje a velké škále dostupných knihoven pro práci s jednotlivými databázovými systémy.

V průběhu testování byly využity 3 databázové technologie (Redis, MySQL, MongoDB). Pro všechny tyto technologie nabízí .NET řadu NuGet[5] balíčků, pomoci kterých lze s každou z těchto databází velmi efektivně komunikovat. V této kapitole budou představeny základní možnosti práce pro databázové systémy Redis a MongoDB. MySQL je z této kapitoly vynecháno vzhledem k tomu, že relačními databázemi nejsou primární náplni této práce a také z toho důvodu, že je práce s relačními databázemi v aplikačním prostředí již základní schopnost téměř

[5]Správce balíčku pro .NET a .NET Core

každého vývojáře. Pro databázový systém MongoDB budou navíc zmíněny ukázky i pro další programovací jazyky, a to z toho důvodu, že se jedná o databázový systém, který má tendence nahradit tradiční relační přístup při návrhu a tvorbě aplikací.

3.2.1 Práce s Redis v klientském prostředí

Redis v platformě .NET

Pro platformu .NET existuje mnoho implementací pro práci s databázovým systémem Redis. Každá implementace nabízí trochu odlišné rozhraní a možnosti. Téměř všechny poskytované knihovny jsou vyvíjeny třetími stranami. Mezi nejpoužívanější balíčky patří csredis, redis-sharp (originální klient pro C#), StackExchange.Redis, **ServiceStack.Redis**. Právě posledně zmíněny balíček je aktuálně nejrozšířenější a poskytuje jedno z nejpropracovanějších rozhraní pro práci s Redis na platformě .NET. Všechny ukázky budou postaveny na rozhraní poskytované právě touto knihovnou. [35]

Samotná knihovna **ServiceStack.Redis** nabízí několik implementací tříd a rozhraní, které pracují se systémem Redis na různých úrovní abstrakce. Nejnižší možná abstrakce, s jakou je možné pracovat je na úrovni bytu. K práci na této úrovni slouží rozhraní `IRedisNativeClient`. Další úroveň abstrakce (zvaná jako textový klient) poskytuje sadu rozhraní `IRedisClient`, `IRedisList`, `IRedisHash`, `IRedisSet` a `IRedisSortedSet`. Všechna tato rozhraní pracují s Redis na textové úrovni. Jinými slovy, všechny metody poskytované těmito rozhraními přijímají jako parametry textové hodnoty, které jsou před odesláním interně převedeny do podoby bytů v kódování UTF-8. Nejvyšší úroveň abstrakce, kterou lze použít je tzv. Typový POCO[6] klient. Jde o stejnou sadu rozhraní jako tomu bylo o úroveň výš akorát s generickým přetížením. Vnitřní logika implementace těchto rozhraní převádí POCO objekty nejprve do formátu JSON a následně do sekvence bytů, které jsou odeslány do Redis.

Hlavním faktorem volby úrovně abstrakce je náročnost pro vlastní implementaci. V tomto ohledu je v C# nejvýhodnější použití typového klienta (nejvyšší úroveň abstrakce). V porovnání s nejnižší úrovní abstrakce není nutné starat se o převod dat do bytů. V případě využití druhé úrovně abstrakce je nutné se starat o převod dat v rámci aplikace, a to z toho důvodu, že metody na této úrovni vracejí data v podobě pole bytů. Důležitou vlastností v C# je také to, že všechny datové typy jsou objektové. Není tedy nutné explicitně vytvářet žádnou POCO třídu v případě, že jsou vkládány pouze hodnoty skalárních datových typu (string, integer, double).

Pro připojení k Redis je nutné vytvořit připojovací řetězec (connection string). Redis podporuje celou řadu syntaxí připojovacího řetězce s celou řadou rozšíření. Základním příkladem připojovacího řetězce je klasická IP adresa s odpovídajícím portem ve tvaru *127.0.0.1:6379*. V případě,

[6] POCO je zkratka pro Plain Old CLR Objekt, jež je označení pro běžnou třídu, která udržuje pouze doménovou logiku a není závislá na žádném frameworku.

že je v Redis povolená autentizace je nutné rozšířit připojovací řetězec o přihlašovací údaje. Výsledný připojovací řetězec vypadat následovně *redis://UzivatelskeJmeno:Heslo@localhost:6380*.

Jakmile jsou staženy všechny potřebné NuGet balíčky a je vytvořen odpovídající připojovací řetězec, je možné se připojit k databázovému systému Redis. V následující ukázce je vytvořeno připojení a v rámci tohoto připojení je do databáze vložena hodnota, která je následně (na základě klíče) získána zpět a poté smazána.

```
using(RedisClient client = new RedisClient(/*ConnectionString*/))
{
   int data = 150;
   string key = "TestNumber";
   bool inserted = client.Add(key, data);
   int insertedData = client.Get<int>(key);
   client.Remove(key);
}
```

Výpis 3: Připojení k Redis pomocí .NET klienta

Příklad výpisu kódu 3 je pouze demonstrativní ukázkou CRUD operací. Úplně stejně bychom mohli pracoval s výše zmíněnými POCO objekty. Vzhledem k tomu, že Redis podporuje spoustu datových struktur (list, množina, seřazený list, hash), tak i tato knihovna nabízí spoustu dalších metod pro práci s těmito datovými typy [36].

3.2.2 Práce s MongoDB v klientském prostředí

MongoDB na platforma .NET

Situace s MongoDB je na platformě .NET lehce odlišná. Existuje pouze jeden oficiální balíček, jež je vyvíjen za podpory samotného MongoDB, Inc. Balíček se jmenuje `MongoDB.Driver` a má závislosti na balíčkách `MongoDB.Bson` a `MongoDB.Libmongocrypt`. První jmenovaný balíček slouží pro reprezentaci dat vhodných pro uložení v MongoDB. Druhý balíček poskytuje implementaci pro zabezpečené připojení a šifrování samotných dat.

Hlavní entitou, která zprostředkovává připojení a přístup k základním funkcionalitám databázového systému MongoDB je třída `MongoClient`. Tato třída poskytuje metodu `GetDatabase`, která reprezentuje databázi s daným jménem. Metoda vrací instanci třídy, která implementuje rozhraní `IMongoDatabase`. Toto rozhraní poskytuje metody pro správu databáze a získání jednotlivých kolekcí. Kolekci lze získat pomocí metody `GetCollection`, které je předáno jméno kolekce. Rozhraní reprezentující kolekci se nazývá `IMongoCollection` a obsahuje všechny metody potřebné pro práci s odpovídající kolekci.

Při získávání kolekce pomocí metody `GetCollection` je nutné specifikovat třídu, která reprezentuje strukturu dat v dané kolekci. To by mohlo vést k domněnce, že pro práci s kolekci je nutné vytvářet POCO objekty, a tedy využívat návrhového vzoru Doménový model. Tyto objekty je samozřejmě možné využívat, nicméně ne vždy je použití těchto objektů žádoucí. Pro tyto případy knihovna `MongoDB.Bson` nabízí třídu `BsonDocument`, která reprezentuje jakoukoliv strukturu dat, využitelnou v databázovém systému MongoDB. Tuto třídu je tedy možné využít jak pro reprezentaci samotných dat v kolekci, tak i pro specifikování filtrovacích a dalších omezení.

Pro specifikování konkrétní instance databázového systému při vytváření připojení je stejně jako v předchozím případě nutné specifikovat připojovací řetězec. Ten je zadáván v podobně *mongodb://UzivatelskeJmeno:Heslo@127.0.0.1:27017/?authSource=admin*. V případě, že není v databázovém serveru nastavena autentizace (ve výchozím stavu vypnutá), je možné využít zkráceného zápisu ve tvaru *mongodb://158.196.98.53:27017*.

V následující ukázce kódu je znázorněno použití CRUD operací pomocí balíčku `MongoDB.Driver` a knihovny `MongoDB.Bson`.

```
BsonDocument document = new BsonDocument
{
   { "Název", "Test" },
   { "Hodnota", 10 },
   { "VnořenýDokument", new BsonDocument { { "VnitřníHodnota", true} } }
}

MongoClient client = new MongoClient(/*ConnectionString*/);
IMongoDatabase database = client.GetDatabase("TestDatabase");
IMongoCollection<BsonDocument> collection =
   database.GetCollection<BsonDocument>("TestCollection");

await collection.InsertOneAsync(document);

IAsyncCursor<BsonDocument> queryResult =
   await collection.FindAsync(new BsonDocument { { "Hodnota", 10 } });

var insertedDocument = await queryResult.FirstOrDefaultAsync();

await collection.DeleteOneAsync(new BsonDocument { { "Hodnota", 10 } });
```

Výpis 4: Připojení k MongoDB pomocí .NET klienta (BsonDocument)

Z uvedené ukázky si lze všimnout, že je práce s knihovnou poměrně přímočará a že definice dokumentu pomocí třídy `BsonDocument` je velice intuitivní. V praxi ovšem existuje daleko více případů, kdy je využit návrhový vzor Doménový model a z tohoto důvodu jsou daleko častěji pro definici dokumentu využívány POCO objekty tak, jak je to znázorněno v ukázce 5.

```
var product = new Product()
{
   Name = "Test",
   Price = 214.65,
   Description = "Testovací produkt"
};

MongoClient client = new MongoClient(/*ConnectionString*/);
IMongoDatabase database = client.GetDatabase("TestDatabase");
IMongoCollection<Product> productCollection =
   database.GetCollection<Product>("Product");

await productCollection.InsertOneAsync(product);

IAsyncCursor<Product> queryResult =
   await productCollection.FindAsync(x => x.Name.Equals("Test"));

var insertedDocument = await queryResult.FirstOrDefaultAsync();

await productCollection.DeleteOneAsync(x => x.Name.Equals("Test"));
```

Výpis 5: Připojení k MongoDB pomocí .NET klienta (POCO objekt)

Při porovnání obou ukázek zjistíme, že jsou téměř totožné. Hlavním rozdílem je využití lambda výrazů při definici filtrů u operace `find` a `delete`. Je také důležité si uvědomit, že i přes velkou podobnost s některými ORM frameworky jako je například EntityFramework pro relační databáze, se jedná pouze o knihovnu, a nikoliv o framework. V prostředí .NET by se tedy dala tato knihovna přirovnat spíše ke knihovně `SqlClient`. Z tohoto plynou určitá omezení týkající se například způsobu mapování podřízených objektů.

Možnosti nabízející tato knihovna jsou velice rozsáhlé. Rozhraní `IMongoClient` nabízí metodu `StartSession`. Tato metoda je důležitá proto, protože samotný databázový systém MongoDB (do verze 4.2) neobsahuje funkcionalitu transakčního zpracování pro operace nad více dokumenty. V případě, že chceme tuto funkcionalitu využít, musíme využít toto programové API.

Další důležité metody jsou poskytovány převážně rozhraním `IMongoCollection`. Některé důležité metody poskytované tímto rozhraním jsou uvedeny v příloze D.

MongoDB v programovacím jazyce Java

Stejně jako v případě platformy .NET je nutné si pro práci s MongoDB stáhnout odpovídající sadu knihoven. V prostředí .NET byla tato situace velmi zjednodušená díky balíčkovacímu systému NuGet. V programovacím jazyce Java je situace o něco komplikovanější. Existují 3 základní způsoby, kterými je možné přidat potřebné knihovny k aplikaci. Prvním a základním způsobem je stažení knihovny a její následné nareferencování do aplikace. Tento způsob je však zastaraly a v dnešní době je doporučováno použití jednoho z nástrojů pro správu, řízení a automatizaci sestavování programu. Mezi dvě nejrozšířenější platformy patří Maven a Gradle. Pro následující ukázku bude tedy použit jeden z těchto doporučených frameworků, a to konkrétně Maven.

Je také důležité si uvědomit, že díky použití balíčkovacího systému NuGet bylo téměř jisté, že při získávání balíčku byla použita nejaktuálnější verze knihovny. Systém Maven neudržuje žádnou databázi verzí balíčku jako NuGet. Stará se pouze o získání balíčků podle definice v sekci *Dependencies*. Je tedy pouze na programátorovi, aby si pohlídat stažení správného balíčku.

K tomu, abychom byli schopni komunikovat s MongoDB z aplikace v programovacím jazyce Java, je nutné do aplikace přidat ovladač pro MongoDB. Způsoby přidání tohoto ovladače byli zmíněny v předchozím odstavci, kde také bylo rozhodnuto, že budou balíčky spravovaný prostřednictvím systému Maven. V následující ukázce je znázorněna struktura, která přidá ovladač pro MongoDB do Java aplikace.

```
<dependencies>
   <dependency>
      <groupId>org.mongodb</groupId>
      <artifactId>mongodb-driver-sync</artifactId>
      <version>4.0.2</version>
   </dependency>
</dependencies>
```

Výpis 6: Nastavení závislosti na ovladači pro MongoDB v systému Maven

Po přidání této konfigurace do konfiguračního souboru systému Maven (pom.xml) je možné využít téměř všech možností databázového systému MongoDB. V předchozí kapitole bylo zmíněno, že ovladače pro většinu programovacích jazyků jsou vyvíjeny pod dohledem organizace, která zodpovídá také za vývoj samotného databázového systému. I přesto jsou implementace pro určité jazyky lehce odlišné. V následující ukázce kódu je ukázán způsob, jakým se připojit k databázovému systému MongoDB a provést CRUD operace nad jednou kolekcí. Příklad je

totožný s příkladem, který byl ukázán pro platformu .NET.

```
ConnectionString connString = new ConnectionString(/*ConnectionString*/);
MongoClientSettings setting = MongoClientSettings.builder()
   .applyConnectionString(connString)
   .build();

MongoClient mongoClient = MongoClients.create(setting);

MongoDatabase database = mongoClient.getDatabase("TestDatabase");
MongoCollection<Document> collection =
           database.getCollection("TestCollection");

Document document = new Document()
   .append("Název", "Test")
   .append("Hodnota", 10)
   .append("VnořenýDokument", new Document().append("VnitřníHodnota", true));

collection.insertOne(document);

FindIterable findIterable =
           collection.find(new Document().append("Hodnota", 10));
Document insertedDocument = (Document) findIterable.first();

collection.deleteOne(new Document().append("Hodnota", 10));
```

Výpis 7: Připojení k MongoDB v programovacím jazyce Java

Z ukázky můžeme vidět několik odlišností oproti implementaci v platformě .NET. Pokud pominneme vlastností samotného jazyka, můžeme vidět hlavní odlišnost při specifikaci parametrů pro připojení, kde se využívá návrhový vzor *Builder*, pomoci kterého se definují všechny parametry pro připojení k databázovému systému MongoDB včetně připojovacího řetězce.

V ukázce pro platformu .NET bylo také ukázáno použití tzv.: POCO objektů. V Javě existuje pro tyto objekty ekvivalent se zkratkou POJO (Plain Old Java Object). Implementace knihovny pro komunikaci s MongoDB v Javě také nabízí možnost využití těchto objektů. Rozdíl oproti platformy .NET je v tom, že zatímco tam byly POCO objekty pomocí reference automaticky převáděny do struktury odpovídající BSON dokumentu, v tomto případě je nutné všechny třídy odpovídající POJO objektům předem registrovat.

MongoDB v programovacím jazyce Python

Ani v případě programovacího jazyka Python není knihovna pro práci s MongoDB součásti standardní instalace a je nutné si ji externě nainstalovat. Opět existuje více způsobu, jak získat knihovny, které lze využít v Python aplikacích. Doporučovanou cestou, jak spravovat balíčky v Python je použitím balíčkovacího správce s názvem **PIP**. Jedná se o podobný nástroj jako je NuGet pro platformu .NET. V rámci tohoto nástroje existuje balíček s názvem *pymongo*, který do Python aplikace přidá všechny potřebné knihovny pro práci s MongoDB. Tento balíček lze získat pomocí následujícího příkazu (ukázka je validní pro Python ve verzi 3.4).

```
pip3 install pymongo
```

Výpis 8: Získání balíčku pymongo

Po úspěšném získáni tohoto balíčku je možné jej naimportovat do projektu a tím využíváte všechny možnosti, které balíček nabízí.

Stejně jako v přechozích příkladech je v následující ukázce kódu vytvořeno připojení k databázovému systému MongoDB a provedení všech základních CRUD operací.

```
import pymongo

client = pymongo.MongoClient('''ConnectionString''')
database = client["TestDatabase"]
collection = database["TestCollection"]
document = {
    "Název": "Test",
    "Hodnota": 10,
    "VnořenýDokument": { "VnitřníHodnota": True}
}
collection.insert_one(document)
insertedDocument = collection.find({"Hodnota": 10})[0]
collection.delete_one({"Hodnota": 10})
```

Výpis 9: Připojení k MongoDB pomocí programovacího jazyka Python

Z ukázky lze vidět, že je práce s MongoDB v Python velice jednoduchá a díky datovým strukturám, které Python poskytuje je mapování mezi MongoDB a samotným Pythonem velice přirozené.

MongoDB v programovacím jazyce PHP

V případě programovacího jazyka PHP je situace velmi podobná jako v případě programovacího

jazyka Java. Existuje zde také několik způsobů, kterými lze získat knihovnu pro komunikaci s MongoDB. Dokumentace MongoDB však doporučuje získáni knihovny pomocí nástroje s názvem **Composer**. Jedná se o nástroj určený pro správu knihoven a jiných zdrojů v PHP. Dalším způsobem, jak získat odpovídající knihovny pro práci s MongoDB v PHP je například stažení GIT repositáře odpovídající knihovny.

Pokud je tedy použit nástroj Composer, je situace se získáním balíčku poměrně jednoduchá. Následující příkaz ukazuje, jak získat všechny potřebné knihovny pro komunikaci s MongoDB v jazyce PHP (je nutné, aby byl příkaz spuštěn z kořenového adresáře projektu). Pod příkazem je také ukázka konfigurace, kterou je nutné přidat do konfiguračního souboru pro PHP (php.ini), který nové rozšíření pro MongoDB a bude tedy možné jej automaticky načítat.

```
//Stažení hnihovny pro komunikaci s MongoDB
composer require mongodb/mongodb

//Nastavení rozšíření pro MongoDB v PHP
extension=php_mongodb.dll
```

Výpis 10: Získání knihovny pro práci s MongoDB v PHP

Tyto kroky stačí k tomu, abychom byli schopni komunikovat s MongoDB. Následující příklad vytvoří připojení k databázovému systému MongoDB a provede CRUD operace stejně jako v předchozích případech.

```
<?php
require_once __DIR__ . "/vendor/autoload.php";

$collection = (new MongoDB\Client(/*ConnectionString*/))
            ->TestDatabase->TestCollection;

$collection->insertOne([
   'Název' => 'Test',
   'Hodnota' => 10,
   'VnitřníDokument' => ["VnitřníHodnota" => true]
]);

$insertedDocument = $collection->findOne(['Hodnota' => 10]);
$collection->deleteOne(['Hodnota' => 10]);
?>
```

Výpis 11: Připojení k MongoDB pomocí programovacího jazyka PHP

Můžeme vidět, že díky beztypovému charakteru jazyka PHP a Python je vytváření dokumentů velmi podobné tomu, jak by tomu bylo přímo v MongoDB. Pro zjednodušení práce s MongoDB v jednotlivých programovacích jazycích lze využít také některé frameworky pro objektově dokumentové mapování (ODM). Pro PHP existuje například framework s názvem *Doctrine MongoDB Object Document Mapper*. Pro Python je poskytovaný framework *pymodm* a pro Javu zase *Morphia*.

3.3 Práce s nestrukturovanými daty v MySQL

V následujících příkladech bude ukázáno, jak lze pracovat s nestrukturovanými daty v databázovém systému MySQL. První část bude věnována práci s formátem JSON a druhá část bude zaměřena na formát XML.

Důležitou otázkou je, proč bychom potřebovali ukládat nestrukturovaná data v relační databázi, která je navržena tak, aby udržovala data podle předem specifikovaného schématu. Vezměme například následující situaci. Chceme ukládat informace o elektronických přístrojích. Mnoho elektronických přístrojů je velmi podobných a liší se pouze v určitých parametrech a jiné mají zase naprosto odlišné vlastnosti. Jednou z možností, jak takové data ukládat je vytvořit pro každý typ produktu vlastní tabulku. Toto řešení vede k velkému počtu různorodých tabulek a nepraktickému mapování na úrovni aplikace. Druhou možností je vytvořit jednu tabulku a pro každý parametr produktu vytvořit jeden atribut. Toto řešení vede k tomu, že bude existovat tabulka s mnoho sloupci a mnoho těchto sloupců bude obsahovat parametr NULL. Třetí možností je vytvoření jedné tabulky, která bude obsahovat atribut popisující vlastnosti daného produktu. Tento atribut může být prostý řetězec, anebo je možné použít jeden z výše zmíněných formátů a pomocí něj uložit parametry produktu pomocí určité struktury. Výhodou takto strukturovaného uložení parametrů produktu je v jednodušším zpracování dat v aplikaci.

Práce s JSON formátem

V následujících ukázkách budou popsány jednotlivé funkce a prostředky, které nabízí MySQL pro dotazování a manipulaci nad tabulkou s JSON formátem.

Pro získávání jednotlivých hodnot z JSON dokumentu lze využít funkci `JSON_EXTRACT`. Tato funkce je při dotazech velmi využívaná, a proto v MySQL (od verze 5.7.9) existuje tzv.: operátor sloupcové cesty. Zapisuje se pomocí kombinace znaků `->` a vykonává tutéž funkci jako `JSON_EXTRACT`.

Pro orientaci v JSON dokumentu se používá speciální zápis, jehož formát je definován ve standardu SQL/JSON a vychází ze syntaxe programovacího jazyka JavaScript. Tento formát tudíž není specifický pouze pro databázový systém MySQL, ale také pro řadu dalších databázových systémů. Syntaxe tohoto doménového jazyka je poměrně jednoduchá a dovoluje nám efektivně manipulovat s JSON dokumentem.

Základem syntaxe je znak `$`, který reprezentuje uvažovaný JSON dokument. Za tímto znakem následují selektory, které dále specifikují konkrétnější část dokumentu. Pro oddělení jednotlivých objektů se využívá tečkové notace. Pro výběr prvků v poli lze použít index prvku, specifikovaný mezi hranatými uvozovkami. Dále lze použít zástupné znaky jako `*` a `**`. Zástupný znak `*` reprezentuje všechny položky v objektu. Pokud se tento znak použije hned za znakem reprezentující JSON dokument (`$`) pak jsou na výstup předány všechny položky dokumentu. Zatímco zástupný znak `*` vybírá všechny podřízené prvky, při použití znaku `**` říkáme, že chceme vybrat všechny položky s daným jménem ze všech nadřazených prvků.

Ukázkový JSON dokument a výstupy ze všech syntaktických variací z následující tabulky jsou uvedeny v příloze A.

Syntaxe	Popis
atributy->'$'	Výstupem je celý JSON dokument.
atributy->'$.Display'	Výstupem je objekt uložený pod názvem *Display*.
atributy->'$.Display.Rozliseni'	Výstupem je objekt uložený pod názvem *Rozliseni*, který je vnořený do objektu *Display*.
atributy->'$.Periferie'	Výstupem je pole, uložené pod názvem *Periferie*.
atributy->'$.Periferie[0]'	Výstupem je první prvek z pole s názvem *Periferie*.
atributy->'$.*'	Výstupem jsou hodnoty všech položek z celého dokumentu bez názvů položek.
atributy->'$**.Typ'	Vypíše hodnoty všech položek s názvem Typ, který je uložen uvnitř jakéhokoliv objektu.
atributy->'$.Display.*'	Výstupem jsou hodnoty všech položek uvnitř objektu s názvem *Display* (bez názvů položek).

Tabulka 9: Ukázky syntaxe pro výběr prvků v JSON dokumentu

Kromě těchto základních operací pro manipulaci s JSON dokumentem při výpisu existuje v MySQL ještě několik funkcí, které se využívají při sestavování a úpravě JSON dokumentu. Je nutné si uvědomit, že tyto funkce pouze modifikují vstupní řetězec a následně jej vrážejí jako výstup. Neprovádějí tedy žádné DML operace.

Pro přidání elementu do JSON dokumentu lze využít funkci `JSON_INSERT`. Ta jako první parametrem přebírá řetězec nebo atribut s JSON dokumentem a za ním následují vždy dvojice parametrů reprezentující cestu, kam má být daná hodnota uložena (včetně nového názvů klíče), a danou hodnotu. Následující SQL příkaz vloží, do již existujícího dokumentu, dva nové elementy (*Velikost*, *Vodeodolny*). Oba tyto elementy jsou vkládány do kořene dokumentu. Samozřejmě ale

může být specifikovaná libovolná validní cesta podle některé z výše zmíněných syntaxí (cesta nesmí obsahovat zástupné znaky * a **).

```
UPDATE Katalog SET atributy =
   JSON_INSERT(atributy, '$.Velikost', '20', '$.Vodeodolny', 'false')
WHERE Id = 2;
```

Výpis 12: Ukázka syntaxe funkce JSON_INSERT

Jak lze vidět z ukázky, tak funkce `JSON_INSERT` neprovádí samotné vkládání záznamu do sloupce s JSON dokumentem, ale pouze modifikuje předaný řetězec. K vložení nových elementů, do již existujícího dokumentu, musíme stále využít klasickou DML operaci UPDATE.

Pokud v předchozím příkladě specifikujeme klíč, který již v daném dokumentu existuje, daný klíč s jeho hodnotou budou ignorovány a operace se korektně dokončí. Pomocí této funkce tedy není možné provádět aktualizace již existujících elementů. K tomuto účelu nám MySQL nabízí dvě další funkce `JSON_REPLACE` a `JSON_SET`. Obě tyto funkce přebírají stejnou množinu parametrů jako funkce `JSON_INSERT`. Jediným rozdílem mezi těmito funkcemi je v tom, že funkce `JSON_REPLACE` neprovede žádnou změnu v dokumentu, pokud specifikovaná cesta není nalezena, zatímco funkce `JSON_SET` v takovém případě vytvoří nový element.

```
UPDATE Katalog SET atributy = JSON_REPLACE(atributy, '$.JeOhebny', 'true')
WHERE Id = 2;
UPDATE Katalog SET atributy = JSON_SET(atributy, '$.JeOhebny', 'false')
WHERE Id = 2;
```

Výpis 13: Ukázka syntaxe funkce JSON_REPLACE a JSON_SET

Oba tyto SQL příkazy jsou korektní (při jejich spuštění neobdržíme žádnou chybu) a jejich účelem je aktualizace JSON dokumentu. První příkaz však neprovede v databázi žádné změny, protože v dokumentu ještě neexistuje element s názvem *JeOhebny*. Druhý příkaz již svou funkci vykoná a přidá nový element do dokumentu. Pokud by byly tyto příkazy spuštěny v opačném pořadí, výsledná hodnota elementu *JeObehny* by byla *true*.

Poslední klasickou DML operací je smazání. Pro tento účel existuje funkce `JSON_REMOVE`. Prvním parametrem funkce je opět řetězec s daným dokumentem a za tímto parametrem následuje množina parametrů, reprezentující cestu odstraňovaných elementů. Pro odstranění elementu *Velikost* lze využít následující SQL příkaz.

```
UPDATE Katalog SET atributy = JSON_REMOVE(atributy, '$.Velikost') WHERE Id = 2;
```

Výpis 14: Ukázka syntaxe funkce JSON_REPLACE a JSON_SET

Ve všech prozatím zmíněných ukázkách byly jako hodnoty ukládány pouze skalární veličiny. Ve většině případů však chceme využít plného potenciálu JSON formátu a ukládat data i v komplexnějších datových formátech jako je objekt nebo pole. I pro tyto případy MySQL obsahuje několik užitečných funkcí.

Pro přidání nového pole do JSON dokumentu lze využít kombinaci funkcí `JSON_INSERT` a `JSON_ARRAY`. Funkce `JSON_ARRAY` vytvoří nové pole a pomocí funkce `JSON_INSERT` jej vložíme do již existujícího dokumentu. Stejně jako v předchozím příkladě dochází pouze k aktualizaci existujícího JSON dokumentu uloženého v jednom sloupci tabulky. Z toho důvodu musíme použít funkci `UPDATE`, která aktualizaci provede. Následující příklad ukazuje, jak lze přidat nové pole do JSON dokumentu.

```
UPDATE Katalog SET atributy =
   JSON_INSERT(atributy, '$.Typ', JSON_ARRAY("A", "B", "C"))
WHERE Id = 2;
```

Výpis 15: Ukázka syntaxe funkce JSON_ARRAY

V případě, že potřebujeme aktualizovat pole, existuje v MySQL funkce `JSON_ARRAY_INSERT`. Ta v parametrech přebírá řetězec aktualizované pole, index, na který se bude nová hodnota vkládat a novou hodnotu.

```
UPDATE Katalog SET atributy =
   JSON_REPLACE(atributy, '$.Typ',
      JSON_ARRAY_INSERT(atributy->'$.Typ', '$[0]', 'D'))
WHERE Id = 2;
```

Výpis 16: Ukázka syntaxe funkce JSON_ARRAY_INSERT

Tento SQL příkaz přidá novou hodnotu do pole *Typ*. Funkce JSON_REPLACE je použita proto, že pole již existuje a je proto nutné jej aktualizovat. Jako druhý parametr funkce s názvem `JSON_ARRAY_INSERT` je zadán výraz, který vloží novou hodnotu na první místo v poli. V tomto případě budou tedy všechny hodnoty uložené v poli posunuty o jedno místo doprava. Vložení prvku na konec pole je možné provést pomocí funkce `JSON_LENGTH`, která vrací velikost pole.

```
UPDATE Katalog set atributy =
   JSON_REPLACE(atributy, '$.Typ', JSON_ARRAY_INSERT(atributy->'$.Typ',
      CONCAT("$[", JSON_LENGTH(atributy->'$.Typ'), "]"), "E"))
WHERE Id = 2;
```

Výpis 17: Ukázka syntaxe funkce JSON_ARRAY_INSERT a JSON_LENGTH

Pro aktualizaci prvků v poli se používá funkce `JSON_UPDATE` a pro odstranění prvku v poli `JSON_REPLACE`. do kterých jako selektor vložíme syntaxi založenou na ukázce z tabulky 9, a to konkrétně z příkladu číslo 5.

S objekty v JSON dokumentu lze také pracovat jednoduše, a to použitím funkce `JSON_OBJECT`, která na základě předaných parametrů vytvoří požadovaný objekt. Parametry se stejně jako v případě funkce `JSON_INSERT` vkládají v podobě dvojic, kdy první parametr dvojice představuje klíč a druhý hodnotu.

```
UPDATE Katalog SET atributy =
   JSON_INSERT(atributy, '$.Rozmery', JSON_OBJECT('Sirka', 20, 'Vyska', 10))
WHERE Id = 2;
```

Výpis 18: Ukázka syntaxe funkce JSON_OBJECT

Díky tomu, že všechny funkce pracují nad řetězcem, můžeme všechny tyto funkce kombinovat a vytvářet tak komplexní struktury podle naších potřeb.

Již v kapitole 2.6 bylo řečeno, že nelze vyvářet indexy nad sloupci, které jsou datového typu `json`. Existují však případy, ve kterých je nutné často vyhledávat právě na základě některé hodnoty v JSON dokumentu. V takových případech by mohlo docházet kvůli absenci indexu k podstatnému snížení rychlosti dotazů. K eliminaci těchto případů lze využít také již dříve zmíněné virtuální sloupce. Vytvoření takového sloupce je téměř totožné s vytvořením klasického sloupce s tím rozdílem, že je nutné specifikovat, že jde o generovaný sloupec a jeho datový zdroj. Ukázkový příkaz, který vytvoří virtuální sloupec je znázorněn v ukázce 19.

```
ALTER TABLE Katalog ADD COLUMN TypDisplejeVS VARCHAR(20) GENERATED ALWAYS
AS (atributy ->> '$.Display.Typ');
```

Výpis 19: Ukázka vytvoření virtuálního sloupce v MySQL

S takto vytvořeným sloupec je nyní možné manipulovat stejně jako se standardním sloupcem tabulky. Je tedy možné pomocí klasického příkazu `CREATE INDEX` nad tímto sloupcem vytvořit index, který urychlí vyhledávání na základě atributu, který byl specifikován v cestě při vytváření virtuálního sloupce.

Práce s XML formátem

XML nemá v MySQL tak velkou podporu jako formát JSON. Množství funkcí, které byly nabízený pro formát JSON je v případě XML velice snížen. V základě jsou implementovány dvě základní funkce. První funkcí je `ExtractValue`. Tato funkce slouží pro získání hodnoty z XML

dokumentu na základě specifikované cesty. Druhou funkcí je `UpdateXML`, která zastřešuje veškerou modifikaci XML dokumentu jako je vkládání, aktualizace a mazání elementů.

Obě výše zmíněné funkce pracují na základě cesty specifikované pomocí jazyka XPath, což je také důvod, proč jsou tyto funkce pro práci s XML dokumentem dostačující. V případě JSON je práce s celým dokumentem trochu složitější než u XML. Je to hlavně proto, že JSON může obsahovat složitější struktury jako jsou pole a zanořené dokumenty. V XML je vytvoření těchto struktur samozřejmě také možné. Je však nutné, aby byla zachována stromová struktura dokumentu, a proto je parsování a manipulace s XML jednodušší. Jazyk XPath má podobnou strukturu jako jazyk používaný u JSON. Největší rozdíl je v tom, že XPath používá pro oddělení uzlu lomítko (/) a dokáže vracet pouze skalární hodnoty. Nelze tedy jako hodnotu vracet určitou část stromu což, jak lze vidět v příkladu níže, je jedna z největších nevýhod implementace jazyka XPath v databázovém systému MySQL.

Syntaxe	Popis
/Atributy/Patice	Vybere hodnotu uloženou v elementu *Patice*, který je zanořen do elementu *Atributy*.
/Atributy/Display/@Typ	Vybere hodnotu uloženou v atributu *Typ*, která je součástí elementu *Display* zanořeného do kořenového elementu *Atributy*.
/Atributy/Periferie /Polozka[1]	Vybere první výskyt elementu *Polozka*, jež je sub elementem elementu *Periferie*.
//Polozka	Vybere všechny elementy *Polozka*, bez ohledu na to, kde se nacházejí.
/Atributy/Display/*	Vypíše hodnoty všech elementů vnořených do elementu *Display*.
/Atributy/Periferie /Polozka[last()]	Vybere poslední výskyt element *Polozka*, jež je sub elementem elementu *Periferie*.
/Atributy/Display[@Typ='LCD'] /UhloprickaPalec	Vybere všechny elementy *UhloprickaPalec*, jež jsou vnořeny do elementu *Display*, který má atribut *Typ* a jeho hodnota je „LCD“.

Tabulka 10: Ukázky syntaxe pro výběr prvků v XML dokumentu

V tabulce 10 jsou ukázány příklady nejpoužívanějších syntaxí jazyka XPath. Tento jazyk má mnoho dalších rozšíření, které zde nejsou zmíněny z důvodu, že nejsou podporovány databázovým systémem MySQL nebo je jejich použití nestandardní. Pro příklady byla použit stejná datová struktura jako v příkladech pro formát JSON. Ukázkové výstupy jednotlivých syntaxí jsou uvedeny v příloze B

Pro získání hodnoty z XML dokumenty se tedy využívá funkce `ExtractValue`, která je ukázána v následujícím SQL příkazu. Její použití je velice podobné funkci `JSON_EXTRACT`. V případě této funkce však neexistuje v MySQL žádný zástupný znak. Její použití je naznačeno v následujícím SQL příkaze.

```
SELECT ExtractValue(atributy, '/Atributy/Patice') FROM Katalog;
```

Výpis 20: Ukázka syntaxe funkce ExtractValue

Veškerá ostatní manipulace s XML dokumentem se provádí pomocí funkce `UpdateXML`. Tato funkce přebírá 3 parametry. Prvním parametrem je samotný XML dokument, druhým parametrem je požadovaná cesta do dokumentu a třetím parametrem je element, který bude nahrazovat stávající element podle specifikované cesty.

Funkce primárně slouží pro úpravu XML dokumentu. Aktualizace libovolného uzlu je tedy v celku přímočará. Princip funkce spočívá v tom, že se nahrazuje element specifikovaný cestou jiným elementem. Díky tomuto chování funkce lze jednoduše provést i smazání určitého elementu z dokumentu. Stačí nahradit požadovaný uzel prázdnou hodnotou. Toto chování je však velice problematické v případě, že chceme do existujícího dokumentu vložit nový element. Pokud specifikovaná cesta není v dokumentu nalezena, nad dokumentem se neprovede žádná změna. Pokud v cestě specifikujeme existující element, dojde k jeho přepsání. Následuje ukázka pro všechny tři výše zmíněné operace.

```
-- Aktualizace elementu
UPDATE Katalog SET atributy =
   UpdateXML(atributy, '/Atributy/Patice', '<Patice>Socket 754</Patice>')
WHERE Id = 1;

-- Smazání elementu
UPDATE Katalog SET atributy = UpdateXML(atributy, '/Atributy/Patice', '')
WHERE Id = 1;

-- Přidání elementu
UPDATE Katalog SET atributy =
   UpdateXML(atributy, '/Atributy/PocetJader',
      CONCAT(
         '<PocetJader>',
         ExtractValue(atributy, '/Atributy/PocetJader'),
         '</PocetJader>',
         '<PocetVlaken>2</PocetVlaken>'))
```

```
WHERE Id = 1;
```

Výpis 21: Ukázka syntaxe funkce UpdateXML

Z ukázky lze vidět, že vkládání je velice problematické a v některých případech prakticky nemožné. Jedním z příkladů, kdy bude téměř nemožné tímto způsobem vložit nový prvek do dokumentu bude, když element, do kterého budeme chtít nový element zanořovat bude obsahovat pouze komplexní elementy (elementy, které obsahují další zanořené elementy). V takovém příkladě bychom musel pomocí funkce `CONCAT` (tak jak to bylo naznačeno v ukázce) nahradit jeden ze zanořených dokumentů úplně celý. SQL příkaz by se tak stal velice nepřehledným.

Shrnutí

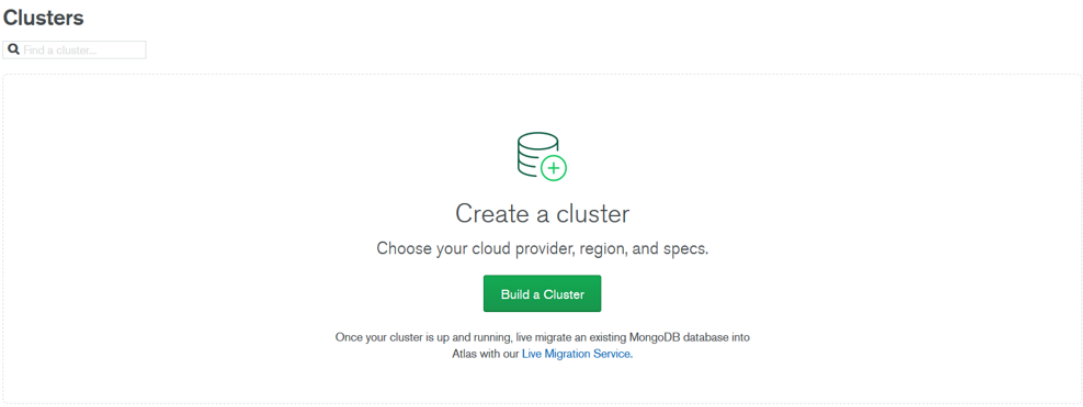
Z výše uvedených příkladů je zřejmé, že je práce s nestrukturovanými daty v MySQL vcelku jednoduchá díky velkému množství poskytovaných funkcí (převážně pro formát JSON). Samozřejmě vzniká otázka, zda je výhodnější manipulovat s těmito formáty na úrovni databáze, nebo v aplikační vrstvě. Obě varianty mají své výhody i nevýhody a je potřeba si uvědomit, jaký význam pro nás tyto data mají. V případě, že je pro nás základem databáze a aplikace je pouze jakousi nadstavbovou vrstvou přístupu, pak je určitě velice výhodně používat uložené procedury a funkce, ve kterých se výše zmíněné funkce uplatní velice dobře. Na druhou stranu, pokud je pro nás databáze pouze nástroj perzistence, pak by určitě stálo za zvážení, zda se raději nepřiklonil ke zpracovávání těchto dat na úrovni aplikace. Samozřejmě lze obě tyto metody kombinovat.

3.4 Ukázka nasazení MongoDB na Azure Cloud

Ukázka nasazení bude probíhat na webovém rozhraní systému MongoDB Atlas (stručně popsaný v teoretické části). Ukázka má za cíl stručně vysvětlit princip nasazení NoSQL databáze (v našem případě MongoDB) na cloudové platformě Azure a vytvořit jednoduchou databázi. Ukázka je svázána s tímto databázovým systémem právě kvůli využití systému MongoDB Atlas, který je určen pouze pro tuto specifickou databázovou technologii. Ostatní databázové systémy mají odlišný způsob nasazení.

Před samotným vytvořením databáze v cloudu je potřeba se zaregistrovat a následně přihlásit do webového rozhraní systému MongoDB Atlas na adrese *https://cloud.mongodb.com/*. Po vyplnění několika formulářů a úspěšném přihlášení nás systém přesměruje na domovskou stránku právě vytvořeného účtu. Zde bychom měli vidět rámeček znázorněn na obrázku 10.

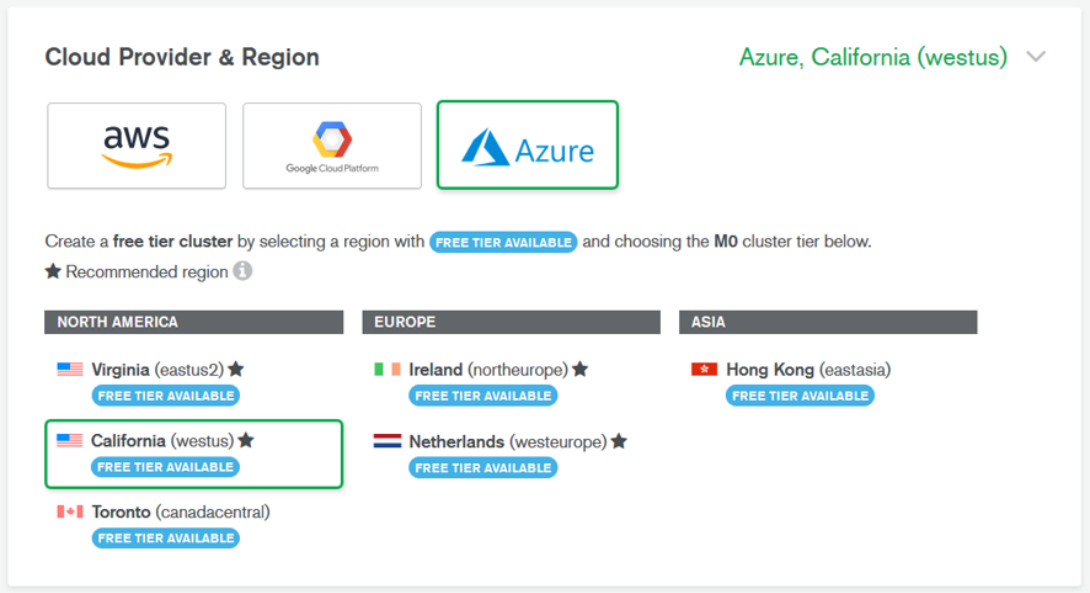
Po kliknutí na zelené tlačítko s textem „Build a Cluster“ budeme přesměrování na stránku, na které jsme požádáni, o výběr typu clusteru, který chceme založit. MongoDB Atlas nabízí možnost vytvoření bezplatného clusteru s označením „Starter Cluster“. Tento typ clusteru má omezené možnost konfigurace a je proto vhodný především pro studijní a testovací účely. Dalšími možnostmi jsou „Single-Region cluster“ a „Multi-Region cluster“. Oba tyto clustery jsou



Obrázek 10: Stránka pro vytvoření clusteru

již placené a poskytují širší možnosti konfigurace a větší výkon. Jsou proto vhodné pro nasazení reálných aplikací. Pro účely této bakalářské práce nám postačí první bezplatná varianta clusteru.

Po vybrání požadovaného typu clusteru budeme přesměrování na další stránku s možností výběru poskytovatele cloudu. Zde jsou v nabídce 3 nejrozšířenější cloudové služby Amazon Web Service, Google Cloud Platform a Microsoft Azure. Každá z těchto služeb nám nabízí možnost vytvoření bezplatného cloudového úložiště v různé lokalitě. Pro naší ukázkovou databázi využijeme platformu Azure v regionu California (výběr je znázorněn na obrázku 11).

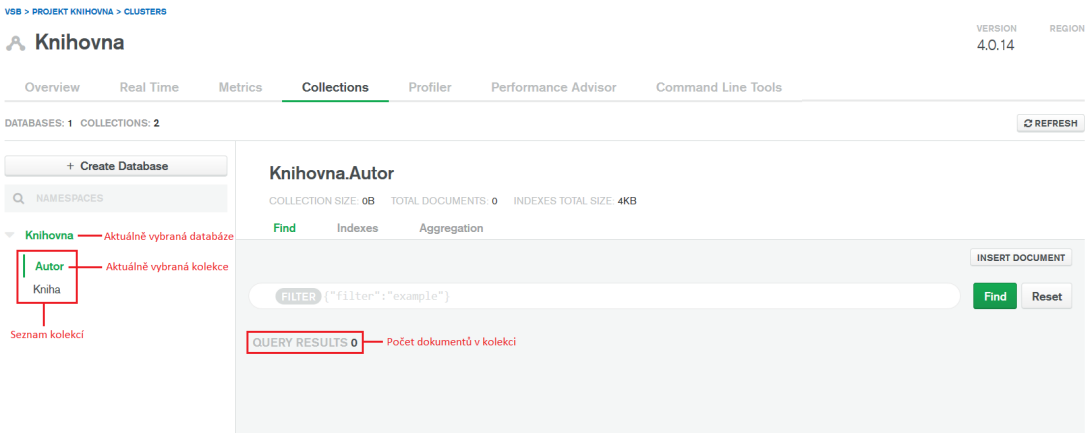


Obrázek 11: Výběr cloudové platformy a regionu

Po výběru platformy a regionu musíme specifikovat jméno clusteru. Zadáme tedy požadované jméno.

Jakmile provedeme tuto sérii kroků, systém nás přesměruje na domovskou obrazovku. Tímto byl náš cluster úspěšně vytvořen a můžeme zde vidět základní informace o něm. V tomto výchozím nastavení se vytvoří jedna primární instance a dvě sekundární. Mezi všemi instancemi probíhá replikace což zvyšuje výkonnost a spolehlivost clusteru.

Nyní lze v clusteru vytvořit databází a uložit v ní data. Samotný MongoDB Atlas poskytuje základní rozhraní pro manipulaci s databázi, kolekcemi a daty. Pro vytvoření nové databáze klikneme na tlačítko „COLLECTIONS“, které se nachází v kartě příslušného clusteru na domovské obrazovce. Po kliknutí nás aplikace přesměruje na stránku se seznamem již vytvořených databází. Pokud ještě není vytvořená žádná databáze, aplikace nám nabídne její vytvořit. Pro vytvoření nové databáze je nutné zadat jméno databáze a jméno kolekce. Po zadání těchto informací bude vytvořena kolekce uvnitř nově vytvořené databáze.



Obrázek 12: Seznam kolekcí v prostředí MongoDB Altas

Nyní je možné do námi vytvořených kolekcí přidávat nové dokumenty. Toto lze zařídit pomocí tlačítka „INSERT DOCUMENT“, které nám otevře nové dialogové okno s možnosti zadání položek dokumentu. V dialogu je předpřipravena položka _id, která slouží pro identifikaci dokumentu. Ostatní položky libovolně přidávat. Při přidání položky je nutné specifikovat název položky, její hodnotu a datový typ. Kromě položek skalárních typů (string, integer, double, . . .), lze vytvářet i vnořené dokumenty a pole.

Kromě manipulace s daty přímo prostřednictvím systému MongoDB Atlas lze pracovat s clusterem i tak, že se na něj vzdáleně připojíme prostřednictvím MongoDB klienta. Jako klienta můžeme použít MongoDB Shell (rozhraní příkazové řádky), konektor pro programovací jazyk, aplikaci MongoDB Compass nebo jednu z mnoha dalších klientských aplikací třetích stran jako jsou například Studio 3T, Robot 3T, NoSQLBooster a NoSQL Manager.

Abychom se mohli připojit do clusteru i pomocí jiných aplikací, musíme specifikovat, z jaké IP adresy je možné přistupovat k danému clusteru. K tomuto nám MongoDB Atlas nabízí tlačítko,

nacházející se na domovské stránce v kartě daného clusteru. Po kliknutí na toto tlačítko se otevře nové dialogové okno.

Při prvotním otevření tohoto dialogu nás systém upozorňuje, že není možné vzdálené připojení a poskytne nám 3 jednoduché kroky pro nastavení firewallu, uživatelských oprávnění a pro získáni připojovacích řetězců pro daný cluster. K tomu, abychom mohli specifikovat zařízení, které budou mít přístup do clusteru je nutné uvést jejich IP adresy do listu povolených adres v prvním bodě. Pro zjednodušení nám systém nabízí možnost automatického vložení aktuální IP adresy. V dalším kroku je nutné specifikovat přihlašovací údaje pro nového uživatele (pokud se jedná o prvního uživatele v systému, tak má automaticky přidělená práva administrátora). Po dokončení těchto kroků se přesuneme na další bod, ve kterém si vybereme způsob připojení. Aplikace nám zde nabídne připojovací řetězce pro námi zvolený způsob připojení. Jakmile získáme tento připojovací řetězec, jsme schopní se do clusteru připojit z jakéhokoliv prostředí, které nabízí možnost připojení do MongoDB databáze.

4 Závěr

S velmi rychle rostoucím výkonem výpočetních zařízení a rostoucí rychlostí počítačové sítě je nutné udržet krok také v odvětví softwaru. Mnoho dnešních systému potřebuje zpracovávat obrovské množství dat během velmi krátkého času. Zde se relační databáze ukazují jako nedostatečné. Je proto často přistupováno ke zvyšování výpočetní kapacity fyzických zařízení, což vede k poměrně velkým finančním nákladům. Možnou alternativou k tomuto přístupu je využití právě některé z NoSQL databázové technologie, které jsou finančně daleko dostupnější, než jsou například systémy od Oracle nebo Microsoft.

Velký rozdíl v rychlosti mezi NoSQL databázemi (MongoDB) a relačními databázemi (MySQL) lze vidět z výše dosažených výsledků výkonnostních testů. Z nich lze vidět, že mají NoSQL systémy velký potenciál pro aplikace, které vyžadují velmi vysoký výkon při práci s daty i za cenu nižší úrovně konzistence dat. Konfigurace jednotlivých databázových systémů byly nastavena tak, aby se ukázaly rozdíly mezi ukládáním dat v podobě normalizovaných a nenormalizovaných dat. Právě nenormalizovaná data jsou doménou NoSQL databázových systému a jednou z hlavního důvodu prosazení NoSQL databází. Výsledky také ukázaly, že jsou NoSQL vhodné i pro situace, kdy nejsou použity jako primární úložiště, a to především pro implementaci cache paměti. Zde se ukázal databázový systém Redis jako velmi vhodnou volbou, která dokáže razantním způsobem navýšit výkon při dotazech prostřednictvím aplikace.

Kromě velkých rychlostí je také velkou dominantou NoSQL databází jednoduchost. Ta se projevuje již při nasazení databázového systému jak na fyzickém zařízení, tak i na cloudové platformě. Velmi jednoduše se s těmito systémy pracuje také prostřednictvím klientů. Téměř každý NoSQL databázový systém nabízí rozhraní pro práci v mnoha programovacích jazycích. Mnoho těchto rozhraní také poskytuje vysokou úroveň abstrakce, která je v dnešní době "enterprise"aplikací velice důležitá. I přes tuto širokou podporu knihoven však stále chybí frameworky pro práci s NoSQL databázemi v mnoha programovacích jazycích.

V práci byla také velká část věnovaná způsobům uložení nestrukturovaných dat v prostředí relačních databázích, a to z toho důvodu, že se tím relační databáze dokáží částečně přiblížit možnostem NoSQL databází. Jde vidět, že je podpora práce s těmito daty v relačních databází velice rozšířená a je tedy možné poměrně efektivně ukládat relační i nestrukturovaná data dohromady. Toto je jedna z výhod, kterou NoSQL databáze nenabízejí. Relační databáze jsou ovšem optimalizované pro normalizovaná data, a proto výkonost dotazů nad nestrukturovanými daty není tak efektivní. Dalším problémem je také to, že spolu s relačními databázemi se často využívají různé frameworky, které práci s funkcemi pro nestrukturované daty ve většině případů nepodporují.

Další důležitou součástí NoSQL systému je také možnost rozsáhlého horizontálního škálování, která tvoří velmi podstatnou výhodu oproti relačním databázím, které tuto možnost nabízí

pouze v omezené míře. Toto odvětví databázových systému zde bylo zmíněno pouze okrajově. Téma horizontálního škálování v NoSQL databázích by mohlo být námětem pro další rozpracování.

Závěrem by se dalo shrnout, že jsou NoSQL databáze vhodný pro všechny případy, kdy potřebujeme velmi vysoký výkon a konzistence dat pro nás není tím nejdůležitějším faktorem. Dalším důvodem, proč se přiklonit k NoSQL databázím spíše, než k relačním databázím je nativní podpora uložení nestrukturovaných dat. Relační databáze bychom na druhou stranu měli upřednostnit tehdy, když potřebujeme mít jistotu, že jakmile dojde jednou k uložení dat, pak jsou data trvale uložena v systému. Toto je nezbytné pro mnoho typů informačních systémů, které vyžadují téměř stoprocentní konzistenci dat.

Literatura

1. EDUCBA. *JSON vs BSON* [online]. 2018. Version 1.1 [cit. 2020-03-22]. Dostupné z: https://www.educba.com/json-vs-bson/.
2. JSON SCHEMA. *The home of JSON Schema* [online]. 2019. Version 1.1 [cit. 2020-03-22]. Dostupné z: https://json-schema.org/.
3. W3C SCHOOL. *DTD Tutorial* [online] [cit. 2020-03-22]. Dostupné z: https://www.w3schools.com/xml/xml_dtd_intro.asp.
4. W3C SCHOOL. *XML Schema tutorial* [online] [cit. 2020-03-22]. Dostupné z: https://www.w3schools.com/xml/schema_intro.asp.
5. HOLUBOVÁ IRENA Minařík Karel, Novák David. *Big Data a NoSQL databáze*. Ed. HICKS, Robert Drew. Grada, 2015.
6. CELKO, Joe. *Joe Celko's Complete Guide to NoSQL: What Every SQL Professional Needs to Know about Non-Relational Databases*. Ed. by HICKS, Robert Drew. Morgen Kaufmann, 2014.
7. BIGDATA DESK. *TOP 12 NoSQL document databases* [online]. 2013 [cit. 2020-03-22]. Dostupné z: https://www.predictiveanalyticstoday.com/top-nosql-document-databases/.
8. AWS, Amazon Web Service. *What Is a Document Database?* [online]. 2020 [cit. 2020-03-22]. Dostupné z: https://aws.amazon.com/nosql/document/.
9. TIMESTORED, Team. *What is a Column-Oriented Database?* [online]. 2016 [cit. 2020-03-22]. Dostupné z: http://www.timestored.com/time-series-data/what-is-a-column-oriented-database.
10. VARGAS, Kathryn. *The Main NoSQL Database Types* [online]. 2020 [cit. 2020-03-22]. Dostupné z: https://studio3t.com/knowledge-base/articles/nosql-database-types/#graph-store.
11. STUDYTONIGHT, Team. *Normalization of database* [online]. 2020 [cit. 2020-03-22]. Dostupné z: https://www.studytonight.com/dbms/database-normalization.php.
12. KIM, Kay. *MongoDB Wire Protocol* [online]. 2008 [cit. 2020-03-23]. Dostupné z: https://docs.mongodb.com/manual/reference/mongodb-wire-protocol/.
13. APACHE SOFTWARE FOUNDATION. *HTTP API Reference* [online]. 2018 [cit. 2020-03-23]. Dostupné z: https://docs.couchdb.org/en/2.2.0/http-api.html.
14. MANSOOR, Umer. *Caching Strategies and How to Choose the Right One* [online]. 2017 [cit. 2020-03-28]. Dostupné z: https://codeahoy.com/2017/08/11/caching-strategies-and-how-to-choose-the-right-one/.

15. ZUBIALEVICH, Dzmitry. *Caching Strategies* [online]. 2018 [cit. 2020-03-28]. Dostupné z: https://zubialevich.blogspot.com/2018/08/caching-strategies.html.

16. SAM KLEINMAN, Kay Kim. *Model One-to-Many Relationships with Document References* [online]. 2008 [cit. 2020-03-23]. Dostupné z: https://docs.mongodb.com/manual/tutorial/model-referenced-one-to-many-relationships-between-documents/.

17. SAM KLEINMAN Kay Kim, Jeff Allen. *Model One-to-Many Relationships with Embedded Documents* [online]. 2008 [cit. 2020-03-23]. Dostupné z: https://docs.mongodb.com/manual/tutorial/model-embedded-one-to-many-relationships-between-documents/.

18. DAVIS, Kyle. *Goodbye Cache: Redis as a Primary Database* [online]. 2020 [cit. 2020-03-23]. Dostupné z: https://redislabs.com/blog/goodbye-cache-redis-as-a-primary-database/.

19. ROBIE, Jonathan. *SQL/XML Tutorial: SQL/XML, XQuery, and Native XML Programming Languages* [online]. 2005 [cit. 2020-03-23]. Dostupné z: http://www.stylusstudio.com/whitepapers/sqlxml.pdf.

20. WINAND, Markus. *What's New in SQL:2016* [online]. 2017 [cit. 2020-03-23]. Dostupné z: https://modern-sql.com/blog/2017-06/whats-new-in-sql-2016.

21. CRAIG GUIER, Gene Milener. *XML Data Type and Columns (SQL Server)* [online]. 2019 [cit. 2020-03-23]. Dostupné z: https://docs.microsoft.com/en-us/sql/relational-databases/xml/xml-data-type-and-columns-sql-server?view=sql-server-ver15.

22. JAKEL, Milan. *Jak na MS SQL – zajímavé možnosti XML* [online]. 2002 [cit. 2020-03-23]. Dostupné z: https://docs.microsoft.com/en-us/sql/relational-databases/xml/xml-data-type-and-columns-sql-server?view=sql-server-ver15.

23. CRAIG GUIER, Gene Milener. *JSON data in SQL Server* [online]. 2019 [cit. 2020-03-23]. Dostupné z: https://docs.microsoft.com/en-us/sql/relational-databases/json/json-data-sql-server?view=sql-server-ver15.

24. ORACLE. *Oracle XML DB in Oracle Database 12c Release 2* [online]. 2017 [cit. 2020-03-23]. Dostupné z: https://www.oracle.com/technetwork/database-features/xmldb/overview/xmldb-twp-12cr1-1964803.pdf.

25. ORACLE. *JSON in Oracle Database* [online]. 2020 [cit. 2020-03-23]. Dostupné z: https://docs.oracle.com/en/database/oracle/oracle-database/19/adjsn/json-in-oracle-database.html#GUID-F6282E67-CBDF-442E-946F-5F781BC14F33.

26. ŠALKO, Michal. *PostgreSQL: XML co s ním 1. část* [online]. 2016 [cit. 2020-03-24]. Dostupné z: https://blog.root.cz/databaze/postgresql-xml-co-s-nim/.

27. POSTGRESQL. *Documentation PostgreSQL: XML Functions* [online]. 2020 [cit. 2020-03-24]. Dostupné z: https://blog.root.cz/databaze/postgresql-xml-co-s-nim/.

28. POSTGRESQL. *Documentation PostgreSQL: JSON type* [online]. 2020 [cit. 2020-03-24]. Dostupné z: https://www.postgresql.org/docs/9.4/datatype-json.html.

29. STEPHENS, Jon. *Using XML in MySQL* [online]. 2010 [cit. 2020-03-24]. Dostupné z: http://download.nust.na/pub6/mysql/tech-resources/articles/xml-in-mysql5.1-6.0.html.

30. RAVAL, Sudip. *Working with MySQL JSON data type* [online]. 2018 [cit. 2020-03-24]. Dostupné z: https://medium.com/aubergine-solutions/working-with-mysql-json-data-type-with-prepared-statements-using-it-in-go-and-resolving-the-15ef14974c48.

31. LORD, Matt. *Indexing JSON documents via Virtual Columns* [online]. 2016 [cit. 2020-03-24]. Dostupné z: https://mysqlserverteam.com/indexing-json-documents-via-virtual-columns/.

32. MICROSOFT. *Bezserverová architektura a aplikace* [online]. 2020 [cit. 2020-03-24]. Dostupné z: https://azure.microsoft.com/cs-cz/overview/serverless-computing/.

33. MONGODB. *MongoDB Atlas documentation* [online]. 2010 [cit. 2020-03-24]. Dostupné z: https://docs.atlas.mongodb.com/.

34. TOM DYKSTRA Chris Ross, Stephen Halter. *Kestrel web server implementation in ASP.NET Core* [online]. 2020 [cit. 2020-03-24]. Dostupné z: https://azure.microsoft.com/cs-cz/overview/serverless-computing/.

35. REDIS. *Redis clients* [online]. 2020 [cit. 2020-03-31]. Dostupné z: https://redis.io/clients#c.

36. BELLOT, Demis. *ServiceStack Redis* [online]. 2020 [cit. 2020-04-07]. Dostupné z: https://github.com/ServiceStack/ServiceStack.Redis.

A Ukázkové výstupy JSON dokumentu


```
[
    {
        "Patice": "socket 478",
        "PocetJader": 1,
        "FrekvenceJadra": "1.6 Ghz"
    },
    {
        "Barvy": ["červená","černá","čirá"],
        "Material": "silikon",
        "CilovaZnacka": "Xiaomi"
    },
    {
        "Je4K": true,
        "Funkce": ["HbbTV","Smart"],
        "Display": {
            "Typ": "LED",
            "Rozliseni": {"Sirka": 3840, "Vyska": 2160},
            "UhlopříčkaCm": 124.6
        },
        "Periferie": ["Wi-Fi","DVB-T tuner","Sluchátka","USB"],
        "Vlastnosti": {"Hmotnost[kg]": 9.2}
    },
    {
        "Je4K": false,
        "Display": {
            "Typ": "LED",
            "Rozliseni": {"Sirka": 1920, "Vyska": 1080},
            "UhlopříčkaCm": 81.28,
            "UhlopříčkaPalec": 32
        },
        "Periferie": ["DVB-T tuner","Sluchátkový výstup","USB"],
        "Vlastnosti": {
            "USB": {"Typ": "USB 3.0","Pocet": 1},
            "Hmotnost[kg]": 6.6,
            "PrumernyPrikonVKw": 41
        }
```

```
    },
    {
        "Je4K": false,
        "Display": {
            "Typ": "LED",
            "Rozliseni": {"Sirka": 1920, "Vyska": 1080},
            "UhloprickaCm": 81.28,
            "UhloprickaPalec": 32
        },
        "Periferie": ["DVB-T tuner"],
        "Vlastnosti": {"Hmotnost[kg]": 6.2}
    }
]
```

Výpis 22: Vzorový JSON dokument

```
[
    null,
    null,
    {
        "Typ": "LED",
        "Rozliseni": {"Sirka": 3840,"Vyska": 2160},
        "UhloprickaPalec": 32,
        "UhlopříčkaCm": 124.6
    },
    {
        "Typ": "LED",
        "Rozliseni": {"Sirka": 1920,"Vyska": 1080},
        "UhloprickaCm": 81.28,
        "UhloprickaPalec": 32
    },
    {
        "Typ": "LED",
        "Rozliseni": {"Sirka": 1920,"Vyska": 1080},
        "UhloprickaCm": 81.28,
        "UhloprickaPalec": 32
    }
]
```

Výpis 23: Výstup příkladu 1 (atributy->'$.Display')

```
[
    null,
    null,
    {"Sirka": 3840, "Vyska": 2160},
    {"Sirka": 1920,"Vyska": 1080},
    {"Sirka": 1920,"Vyska": 1080}
]
```

Výpis 24: Výstup příkladu 2 (atributy->'$.Display.Rozliseni')

```
[
    null,
    null,
    ["Wi-Fi","DVB-T tuner","Sluchátka","USB"],
    ["DVB-T tuner","Sluchátkový výstup","USB"],
    ["DVB-T tuner"]
]
```

Výpis 25: Výstup příkladu 3 (atributy->'$.Periferie')

```
[null, null,"Wi-Fi","DVB-T tuner","DVB-T tuner"]
```

Výpis 26: Výstup příkladu 4 (atributy->'$.Periferie[0]')

```
[
    ["socket 478", 1, "1.6 Ghz"],
    [["červená", "černá", "čirá"], "silikon", "Xiaomi"],
    [true,["HbbTV","Smart"],
        {
            "Typ": "LED",
            "Rozliseni": {"Sirka": 3840,"Vyska": 2160},
            "UhloprickaPalec": 32,
            "UhlopříčkaCm": 124.6
        },
        ["Wi-Fi","DVB-T tuner","Sluchátka","USB"],
        {"Hmotnost[kg]": 9.2}
    ],
    [false,
        {
            "Typ": "LED",
            "Rozliseni": { "Sirka": 1920,"Vyska": 1080},
            "UhloprickaCm": 81.28,
            "UhloprickaPalec": 32
        },
        ["DVB-T tuner","Sluchátkový výstup","USB"],
        {
            "USB": {"Typ": "USB 3.0", "Pocet": 1},
            "Hmotnost[kg]": 6.6,
            "PrumernyPrikonVKw": 41
        }
    ],
    [false,
        {
            "Typ": "LED",
            "Rozliseni": {"Sirka": 1920,"Vyska": 1080},
            "UhloprickaCm": 81.28,
            "UhloprickaPalec": 32
        },
        ["DVB-T tuner"],
        {"Hmotnost[kg]": 6.2}
    ]
]
```

Výpis 27: Výstup příkladu 5 (atributy->'$.*')

```
[null, null, ["LED"], ["LED", "USB 3.0"], ["LED"]]
```

Výpis 28: Výstup příkladu 6 (atributy->'$**.Typ')

```
[
    null,
    null,
    ["LED", {"Sirka": 3840, "Vyska": 2160}, 32, 124.6],
    ["LED",{"Sirka": 1920, "Vyska": 1080},81.28,32],
    ["LED",{"Sirka": 1920,"Vyska": 1080},81.28,32]
]
```

Výpis 29: Výstup příkladu 7 (atributy->'$.Display.*')

B Ukázkové výstupy XML dokumentu

```
<Atributy>
    <FrekvenceJadra>1.6 Ghz</FrekvenceJadra>
    <Patice>socket 478</Patice>
    <PocetJader>1</PocetJader>
    </Atributy>
<Atributy>
    <CilovaZnacka>Xiaomi</CilovaZnacka>
    <Barvy>
        <Polozka>cervena</Polozka>
        <Polozka>cerna</Polozka>
        <Polozka>cira</Polozka>
    </Barvy>
    <Material>silikon</Material>
</Atributy>
<Atributy>
    <Display Typ="LED">
        <Rozliseni Sirka="3840" Vyska="2160" />
        <UhloprickaPalec>32</UhloprickaPalec>
        <UhloprickaCm>124.6</UhloprickaCm>
    </Display>
    <Periferie>
        <Polozka>Wi-Fi</Polozka>
        <Polozka>DVB-T tuner</Polozka>
        <Polozka>Sluchátka</Polozka>
        <Polozka>USB</Polozka>
    </Periferie>
    <Je4K>true</Je4K>
    <Funkce>
        <Polozka>HbbTV</Polozka>
        <Polozka>Smart</Polozka>
    </Funkce>
    <Vlastnosti>
        <Hmotnost>9.2</Hmotnost>
    </Vlastnosti>
</Atributy>
<Atributy>
    <Display Typ="LED">
```

```
        <Rozliseni Sirka="1366" Vyska="768" />
        <UhloprickaPalec>32</UhloprickaPalec>
        <UhloprickaCm>81</UhloprickaCm>
    </Display>
    <Periferie>
        <Polozka>DVB-T tuner</Polozka>
        <Polozka>USB</Polozka>
    </Periferie>
    <Je4K>false</Je4K>
    <Funkce>
    </Funkce>
    <Vlastnosti>
        <Hmotnost>6.6</Hmotnost>
        <USB>
            <Pocet Hodnota="1" />
            <Typ Hodnota="Typ C" />
        </USB>
    </Vlastnosti>
</Atributy>
<Atributy>
    <Display Typ="LED">
        <Rozliseni Sirka="1920" Vyska="1080" />
        <UhloprickaPalec>32</UhloprickaPalec>
        <UhloprickaCm>81.28</UhloprickaCm>
    </Display>
    <Periferie>
        <Polozka>DVB-T tuner</Polozka>
        <Polozka>Sluchatkový vystup</Polozka>
        <Polozka>USB</Polozka>
    </Periferie>
    <Je4K>false</Je4K>
    <Funkce>
    </Funkce>
    <Vlastnosti>
        <Hmotnost>6.6</Hmotnost>
        <USB>
            <Pocet Hodnota="1" />
            <Typ Hodnota="USB 3.0" />
        </USB>
```

```
        <PrumernyPrikonKw>41</PrumernyPrikonKw>
    </Vlastnosti>
</Atributy>
<Atributy>
    <Display Typ="LCD">
        <Rozliseni Sirka="1920" Vyska="1080" />
        <UhloprickaPalec>32</UhloprickaPalec>
        <UhloprickaCm>81.28</UhloprickaCm>
    </Display>
    <Periferie>
        <Polozka>DVB-T tuner</Polozka>
    </Periferie>
    <Je4K>false</Je4K>
    <Funkce>
    </Funkce>
    <Vlastnosti>
        <Hmotnost>6.2</Hmotnost>
    </Vlastnosti>
</Atributy>
```

Výpis 30: Ukázka vzorového XML dokumentu

```
socket 478; null; null; null; null; null
```

Výpis 31: Výstup příkladu 1 (/Atributy/Patice)

```
null; null; LED; LED; LED; LCD
```

Výpis 32: Výstup příkladu 2 (/Atributy/Display/@Typ)

```
null; null; Wi-Fi; DVB-T tuner; DVB-T tuner; DVB-T tuner
```

Výpis 33: Výstup příkladu 3 (/Atributy/Periferie/Polozka[1])

```
cervena cerna cira; Wi-Fi DVB-T tuner Sluchatka USB HbbTV Smart; DVB-T tuner
    USB; DVB-T tuner Sluchatkovy vystup USB; DVB-T tuner
```

Výpis 34: Výstup příkladu 4 (//Polozka)

```
null; null; 32 124.6; 32 81; 32 81.28; 32 81.28
```

Výpis 35: Výstup příkladu 5 (/Atributy/Display/*)

```
null; null; USB, USB, USB, DVB-T tuner
```

Výpis 36: Výstup příkladu 6 (/Atributy/Periferie/Polozka[last()])

```
null; null; null; null; null; 32
```

Výpis 37: Výstup příkladu 7 (/Atributy/Display[@Typ='LCD']/UhloprickaPalec)

C Porovnání způsobu dotazování mezi MongoDb a SQL

SQL	MongoDb	Poznámka
CREATE TABLE uzivatel (id INT NOT NULL AUTO_INCREMENT, jmeno VARCHAR(30), vek NUMBER, status CHAR(1), PRIMARY KEY (id));	db.uzivatel.insertOne({ uzivatel_id: 1, jmeno: "abcd", vek: 20, status: "A" })	Kolekce se automaticky vytvoří při prvním vložení dokumentu. Každý dokument implicitně obsahuje prvek _id, který slouží jako primární klíč.
	db.createCollection("uzivatel")	Kolekci lze také vytvořit explicitně bez vkládáné nového dokumentu.
ALTER TABLE uzivatel ADD datum_prihlaseni DATETIME;	db.uzivatel.updateMany({ }, { $set: { datum_prihlaseni: new Date() } })	Kolekce nevynucují žádné schéma. Proto neexistuje žádná operace na změnu struktury kolekce. Místo tohot se pomocí operace SET se provede změna na úrovni dokumentu.
ALTER TABLE uzivatel DROP COLUMN datum_prihlaseni;	db.uzivatel.updateMany({ }, { $unset: { datum_prihlaseni: "" } })	Odstranění elementu lze provést pomocí operace UNSET na úrovni dokumentu.
CREATE INDEX idx_uzivatel_vek ON uzivatel(vek);	db.uzivatel.createIndex({ vek: 1 })	1 = vzestupný index, -1 = sestupný index
DROP TABLE uzivatel;	db.uzivatel.drop()	
INSERT INTO uzivatel(jmeno, vek, status) VALUES ("abcd", 20, "A");	db.uzivatel.insertOne({ jmeno: "abcd", vek: 20, status: "A" })	Pro mnohonásobné přidávání existuje operace insertMany.
SELECT * FROM uzivatel;	db.uzivatel.find()	
SELECT jmeno, vek FROM uzivatel;	db.uzivatel.find({ }, { jmeno: 1, vek: 1 })	Společně s prvky jmeno a vek se výpíše i prvke _id. Proto aby k tomu nedocházelo, je potřeba do příkazu za prvke vek přidat _id: 0.
SELECT * FROM uzivatel WHERE status = "A";	db.uzivatel.find({ status: "A" })	
SELECT * FROM uzivatel WHERE status != "A";	db.uzivatel.find({ status: { $ne: "A" } })	
SELECT * FROM uzivatel WHERE status = "A" AND vek = 20;	db.uzivatel.find({ status: "A", vek: 50 })	
SELECT * FROM uzivatel WHERE status = "A" OR vek = 20;	db.uzivatel.find({ or: [{ status: "A" }, { vek: 20 }]	
SELECT * FROM uzivatel WHERE vek > 15;	db.uzivatel.find({ vek: { $gt: 15 } })	Podobné funkce: $gte - " >= " $lt - " < " $lte - " <= " $exists - " !=null "
SELECT * FROM uzivatel WHERE jmeno LIKE '%cd%'	db.uzivatel.find({ jmeno: /cd/ })	Pomoci této syntaxe lze filtrovat na základě jakéhokoliv validního regulárního výrazu.
	db.uzivatel.find({ jmeno: { regex: /cd/ })	
SELECT * FROM uzivatel WHERE status = "A" ORDER BY vek ASC;	db.uzivatel.find({ status: "A" }).sort({ vek: 1 })	Pro sestupné třídění se použije záporná hodnota (-1) u definice třídění.
SELECT COUNT(*) FROM uzivatel;	db.uzivatel.count()	
	db.uzivatel.find().count()	
SELECT DISTINCT(jmeno) FROM uzivatel;	db.uzivatel.aggregate([{ $group: { _id: "$jmeno" } }])	
	db.uzivatel.distinct("jmeno")	

Obrázek 13: Rozdíl v dotazování mezi SQL a MongoDB 1

SELECT * FROM uzivatel LIMIT 1;	db.uzivatel.findOne() db.uzivatel.find().limit(1)	
SELECT * FROM uzivatel LIMIT 5 SKIP 10;	db.uzivatel.find().limit(5).skip(10)	
UPDATE uzivatel SET status = "C" WHERE vek > 15;	db.uzivatel.updateMany({ vek: { $gt: 15 } }, { $set: { status: "C" } })	
UPDATE uzivatel SET vek = vek + 1 WHERE status = "A";	db.uzivatel.updateMany({ status: "A" }, { $inc: { vek: 1 } })	
DELETE FROM uzivatel WHERE vek > 15;	db.uzivatel.deleteMany({ $gt: { vek: 15 } })	
DELETE FROM uzivatel;	db.uzivatel.deleteMany({ })	

Obrázek 14: Rozdíl v dotazování mezi SQL a MongoDB 2

D Metody pro práci s MongoDB v prostředí .NET

Metoda	Popis
DeleteOne	Vymaže jeden dokument.
DeleteMany	Vymaže jeden až N dokumentů.
InsertOne	Vloží jeden dokument.
InsertMany	Vloží jeden až N dokumentů.
UpdateOne	Aktualizuje jeden dokument.
UpdateMany	Aktualizuje jeden až N dokumentů.
FindAsync	Asynchronní metoda pro získání data z kolekce.
FindSync	Synchronní verze předchozí metody.
FindOneAndDelete	Vyhledá jeden dokument a automaticky jej smaže.
FindOneAndReplace	Vyhledá jeden dokument a automaticky jej nahradí.
FindOneAndUpdate	Vyhledá jeden dokument a automaticky jej aktualizuje.
BulkWrite	Vykonává hromadnou operaci zápisu.
Count	Vrátí počet dokumentů v kolekci.
Distinct	Vrátí pouze jedinečné hodnoty specifikovaného pole.

Tabulka 11: Seznam užitečných metod poskytované rozhraním IMongoCollection